**IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
ON APPEAL FROM THE EXAMINER TO THE BOARD
OF PATENT APPEALS AND INTERFERENCES**

In re Application of:	Antony John Rogers et al.
Serial No.:	09/905,532
Filing Date:	July 14, 2001
Group Art Unit:	2437
Confirmation No.:	3485
Examiner:	Michael J. Pyzocha
Title:	DETECTION OF VIRAL CODE USING EMULATION OF OPERATING SYSTEM FUNCTIONS

Commissioner for Patents
P.O. Box 1450
Alexandria, Virginia 22313-1450

Dear Sir:

APPEAL BRIEF

Appellants have appealed to the Board of Patent Appeals and Interferences (the "Board") from the Final Office Action dated February 10, 2009 (the "Final Office Action"), finally rejecting Claims 1, 4, 8-16 and 20-23. Appellants filed a Notice of Appeal and Pre-Appeal Brief on July 9, 2009 with the statutory fee of $540.00. This Appeal Brief is filed in response to the Final Office Action and the Notice of Panel Decision from Pre-Appeal Brief Review dated September 4, 2009.

REAL PARTY IN INTEREST

This Application is currently owned by Computer Associates Think, Inc. as indicated by an assignment recorded on January 29, 2002 from inventors Antony John Rogers, Trevor Yann, and Myles Jordan to Computer Associates Think, Inc., in the Assignment Records of the PTO at Reel 012536, Frame 0644 (4 pages).

RELATED APPEALS AND INTERFERENCES

To the knowledge of Appellants' counsel, there are no known appeals, interferences, or judicial proceedings that are related to or will directly affect, be directly affected by, or have a bearing on the Board's decision regarding this Appeal.

STATUS OF CLAIMS

Claims 1, 4, 8-16 and 20-23 are pending and stand rejected pursuant to a Final Office Action dated February 10, 2009 ("*Final Office Action*") and a Notice of Panel Decision from Pre-Appeal Brief Review dated September 4, 2009 ("*Panel Decision*"). Claims 6, 7, 18, and 19 were previously cancelled in a Response Pursuant to 37 C.F.R. § 1.111 submitted by Appellants on August 22, 2005. Claims 2 and 3 were previously cancelled in a Response Pursuant to 37 C.F.R. § 1.111 submitted by Appellants on January 30, 2006. Claims 5 and 17 were previously cancelled in a Response Pursuant to 37 C.F.R. § 1.111 submitted by Appellants on October 6, 2006

Specifically, the *Final Office Action* includes the following rejections:

1. Claims 1, 4, 8-16, 20, 22 and 23 were rejected under 35 U.S.C. § 103(a) as being unpatentable over U.S. Patent No. 6,192,512 issued to Chess ("*Chess*") in view of U.S. Patent No. 6,851,057 issued to Nachenberg ("*Nachenberg"*).

2. Claim 21 is rejected under 35 U.S.C. 103(a) as being unpatentable over the modified Chess and Nachenberg system as applied to claim 1 above, and further in view of U.S. Patent No. 5,398,196 to Chambers ("*Chambers*").

For the reasons discussed below, Appellant respectfully submits that the rejections of Claims 1, 4, 8-16 and 20-23 are improper and should be reversed by the Board. Accordingly, Appellants presents Claims 1, 4, 8-16 and 20-23 for Appeal. All pending claims are shown in Appendix A, attached hereto.

STATUS OF AMENDMENTS

All amendments submitted by Appellants have been entered by the Examiner prior to the mailing of the Final Office Action.

SUMMARY OF CLAIMED SUBJECT MATTER

In certain embodiments, as illustrated in FIGURE 1, the present invention comprises a computer system or computer 40 on which computer executable code may execute and/or reside (and which thus may be a target the viral code). Computer system 40 comprises a processor 41, memory 42, hard disk 43, removable storage drive 44 (for reading/accessing removable storage media, such as floppy disks, CDs, DVDs, etc.), display 46, I/O devices 47 (for example, keyboard, mouse, microphone, speaker, etc.), and a wired or wireless connection to a network 48. The network can be, for example, a LAN, a WAN, an intranet, an extranet, the Internet, and/or any combinations of such networks. *See* Spec. at 5:9-16.

Computer 40 may be any of the computing devices/systems known in the art, such as, for example, a personal computer, a laptop, a workstation computer, a mainframe computer, a personal digital assistant (PDA), etc. (also referred to herein either separately or collectively as "computing device", "computer", "computer system" or "computing system"). Subject files may reside on/in, for example, hard disk 43 and/or a removable storage medium that may be read/accessed through removable storage drive 44. Also, the subject computer executable code may be downloaded to the computer system or computer through network 48. *See* Spec. at 5:16-23.

FIGURE 2 illustrates an apparatus for detecting viral code that uses calls to an operating system to damage computer systems, computers and/or computer files. The apparatus 30 includes CPU emulator 36, memory manager component 31 and monitor component 32. Optionally, the apparatus 30 may also include an auxiliary component 33 and analyzer component 34. *See* Spec. at 5:24-29.

As shown by FIGURES 2 and 3, one embodiment according to the present disclosure provides a method for detecting viral code that uses calls to an operating system to damage computer systems, computers and/or computer files will be described. In this embodiment, an artificial memory region 35 spanning one or more components of the operating system is created by memory manager component 31 (step 21). Once the artificial memory region is created, execution of computer executable code in a subject file is emulated by the CPU emulator 36 (step 22). Attempts by the emulated computer executable code to scan the newly

created artificial memory region are detected by monitor component 32 (step 23), wherein any access to the newly created artificial memory region is immediately suspicious, because a legitimate program has no need to access the artificial memory region. If the monitor component 32 detects attempts to access the newly created artificial memory region, the computer executable code is deemed to be viral. *See* Spec. at 5:30-6:12.

Additionally, an embodiment for detecting viral code that uses calls to an operating system to propagate to or damage computer systems, computers and/or computer files is described with reference to FIGURES 2 and 4. Initially, the memory manager component 31 creates an artificial memory region 35 that spans an export table of one or more major components of the operating system (step 11). The CPU emulator 36 emulates execution of computer executable code in a subject file (step 12), and the monitor component 32 detects when emulated code attempts to scan the new artificial memory region (step 13). *See* Spec. at 6:13-20. The auxiliary component 33 determines the operating system call that is being accessed by the emulated code (step 14), and emulates the functionality of the operating system call without halting execution of the code (step 15). Since emulation of the code continues, viral code which initially make one or more innocuous operating system calls may be detected at a later point. *See* Spec. at 6:20-25.

Analyzer component 34 then monitors the operating system call to determine whether the emulated code is viral (step 16). As an example, the analyzer component 34 may monitor access to the artificially created memory region 35 for various suspect viral characteristics, including looping. *See* Spec. at 6: 25-28.

A direct triggering mechanism or an indirect triggering mechanism may be used by the apparatus 30 to detect access by the emulated code to an export table of relevant operating system components. It should be noted that the operation of the apparatus is independent of the operating system being monitored. *See* Spec. at 6:29-7:2.

The functionality of monitoring artificially created memory regions permits the apparatus to monitor operating systems that map key operating system functionality into the memory space. The functions monitored include, but are not limited to, file, process and

module handling. The apparatus may be adapted with other functionalities to detect suspicious (e.g., viral code) calls to operating systems which use other mechanisms (such as system traps) to access operating system calls. *See* Spec. at 7:3-8.

An exemplary embodiment of the present disclosure as implemented on a Win32 platform, which includes Windows95, Windows98, Windows2000, Windows NT, Windows CE and Windows ME will be described below. Each of these operating systems provides key operating system functions by way of entry points in the Kerne132 DLL. *See* Spec. at 7:9-13.

To detect attempts to access the functionalities in the Kernel32 DLL, a region covering the entire export table in the emulated Kernel32 DLL is added to the memory manager. By adding an artificial memory region, any access to the artificial memory region is immediately suspicious, because a legitimate program has no need to access the artificial memory region. Operating system functions can and should be accessed via the program's import table, or through use of the GetProcAddress system call. *See* Spec. at 7:14-19.

In particular embodiments, detecting access to the export table is the first step. The next step is determining the system call that the code attempted to locate. A custom version of the export table, with pre-selected or predetermined values for the entry points, may be provided. This simplifies the process of converting calls to these operating system functions into identified system calls, and allows the emulator to simulate the effect of calling those functions. This part of the emulation may be driven by data tables, which can be replaced easily should it prove necessary to emulate additional calls for combating new viruses. *See* Spec. at 7:20-26.

The technique described above provides a mechanism for detecting unusual access to functions in dynamically linked libraries. The libraries are mapped into an address space of an application, and are usually accessed through information generated by a linker when an executable is created. A program loader initializes a jump table to refer to the location where each function has been mapped into the application address space. Access to these dynamically linked functions is normally done via the jump table, but it is possible to directly

call functions if it can be determined where in the address space the library functions have been mapped. *See* Spec. at 7:27-8:4.

Many of the documented operating systems that function on the Windows platforms (from Windows 1.0 up to and including Windows 2000 and Windows ME) are accessed through dynamically linked libraries. Therefore, it is possible to detect operating system functions that are called in an unusual fashion. Many non-operating system functions also are accessed through dynamically linked libraries, and the method described above also would allow us to check for unusual access to these functions. In addition, Macintosh platforms support dynamically linked libraries. Therefore, the method described above for detecting viral code that uses calls to an operating system to propagate also may be adapted, as would be apparent to persons of ordinary skill in the art after reading this disclosure, the drawings and the appended claims, to apply to such platforms. In addition, modem Unix-based platforms support and use dynamic libraries, and so the method described above may be adapted for detecting unusual access to functions in dynamic libraries on such platforms. Most flavors of the Unix operating systems do not access base operating system functions through dynamic libraries, but often the runtime library is a dynamically linked library. The method described above may be adapted to detect unusual access to runtime library functions. *See* Spec. at 8:5-20.

The apparatus and methods described above may be embodied in a computer program (or some unit of code) stored on/in computer readable medium, such as memory, hard drive or removable storage media. The apparatus and methods also may be in the form of a computer data signal, in one or more segments, embodied in a transmission medium, such as the Internet, an intranet, or another (wired or wireless) transmission medium. The present disclosure also encompasses, of course, execution of the computer program stored on/in a program storage device in a computing device/system, such as, for example, shown in FIGURE 1. *See* Spec. at 8:21-28.

Regarding the independent claims under Appeal, Appellants provide the following concise explanation of the subject matter recited in the claims. For brevity, Appellants do not necessarily identify every portion of the Specification and drawings relevant to the recited

claim elements. Additionally, this explanation should not be used to limit Appellants' claims as it is intended to assist the Board in considering the Appeal of this Application.

For example, independent Claim 1 recites the following:

> A method of detecting viral code in subject files, comprising:
>
> creating an artificial memory region spanning one or more components of the operating system, wherein the artificial memory region is associated with an export table of a dynamically-linked library (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:6-8, 3:17-19, 3:22-24, 4:2-4; 5:30-6:7, 6:13-18, 7:14-19);
>
> emulating execution of at least a portion of computer executable code in a subject file (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:2-5, 3:13-15, 3:21-25, 6:4-6, 6:18-24, 7:20-26);
>
> monitoring attempts by the emulated computer executable code to access the artificial memory region (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:10-12, 3:13-18, 3:21-22, 3:25-26, 6:6-12, 6:19-20, 6:25-28, 7:3-8, 7:14-8:20);
>
> in response to detecting an attempt to access the artificial memory region, determining an export table entry in the export table of the dynamically-linked library that is associated with the attempt to access the artificial memory region (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:10-12, 3:13-19, 6:25-7:8, 7:12-20, 7:27-8:20); and
>
> determining based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral (*see, e.g.,* FIGURES 2 and 4; Spec. at 3:10-12, 3:20-4:4, 6:16-18, 6:29-7:8, 7:20-26, 8:2-4, 8:7-20).

Claim 10 recites:

> A program storage device readable by a machine, tangibly embodying a program of instructions executable by the machine to perform method steps for detecting viral code in subject files, the method steps comprising:
>
> creating an artificial memory region spanning one or more components of the operating system, wherein the artificial memory region is associated with an export table of a dynamically-linked library (*see, e.g.,* FIGURES 2, 3, and 4;-Spec. at 3:6-8, 3:17-19, 3:22-24, 4:2-4; 5:30-6:7, 6:13-18, 7:14-19);
>
> emulating execution of at least a portion of computer executable code in a subject file (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:2-5, 3:13-15, 3:21-25, 6:4-6, 6:18-24, 7:20-26);
>
> monitoring attempts by the emulated computer executable code to access the artificial memory region (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:10-12, 3:13-18, 3:21-22, 3:25-26, 6:6-12, 6:19-20, 6:25-28, 7:3-8, 7:14-8:20);
>
> in response to detecting an attempt to access the artificial memory region, determining an export table entry in the export table of the dynamically-linked library that is associated with the attempt to access the artificial memory region (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:10-12, 3:13-19, 6:25-7:8, 7:12-20, 7:27-8:20); and

determining based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral (*see, e.g.,* FIGURES 2 and 4; Spec. at 3:10-12, 3:20-4:4, 6:16-18, 6:29-7:8, 7:20-26, 8:2-4, 8:7-20).

Claim 11 recites:

A computer system, comprising:

a processor (*see, e.g.,* FIGURES 1 and 2, Spec. at 5:9-23); and

a program storage device readable by the computer systems, tangibly embodying a program of instructions executable by the processor to perform method steps for detecting viral code in subject files (*see, e.g.,* FIGURES 1 and 2, Spec. at 5:9-23), the method comprising:

creating an artificial memory region spanning one or more components of the operating system, wherein the artificial memory region is associated with an export table of a dynamically-linked library (*see, e.g.,* FIGURES 2, 3, and 4;-Spec. at 3:6-8, 3:17-19, 3:22-24, 4:2-4; 5:30-6:7, 6:13-18, 7:14-19);

emulating execution of at least a portion of computer executable code in a subject file (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:2-5, 3:13-15, 3:21-25, 6:4-6, 6:18-24, 7:20-26);

monitoring attempts by the emulated computer executable code to access the artificial memory region (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:10-12, 3:13-19, 6:25-7:8, 7:12-20, 7:27-8:20); and

determining based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral (*see, e.g.,* FIGURES 2 and 4; Spec. at 3:10-12, 3:20-4:4, 6:16-18, 6:29-7:8, 7:20-26, 8:2-4, 8:7-20).

Claim 12 recites:

A computer data signal embodied in a computer-readable medium which embodies instructions executable by a computer for detecting in a subject file viral code that uses calls to an operating system, the signal comprising:

a first segment comprising CPU emulator code, wherein the CPU emulator code emulates execution of at least a portion of computer executable code in the subject file (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:2-5, 3:13-15, 3:21-25, 6:4-6, 6:18-24, 7:20-26);

a second segment comprising memory manager code, wherein the memory manager code creates an artificial memory region spanning components of the operating system, wherein the artificial memory region is associated with an export table of a dynamically-linked library (*see, e.g.,* FIGURES 2, 3, and 4;-Spec. at 3:6-8, 3:17-19, 3:22-24, 4:2-4; 5:30-6:7, 6:13-18, 7:14-19); and

a third segment comprising monitor code, wherein the monitor code:

monitors attempts by the emulated computer executable code to access the artificial memory region (*see, e.g.,* FIGURES 2, 3, and 4;

Spec. at 3:10-12, 3:13-18, 3:21-22, 3:25-26, 6:6-12, 6:19-20, 6:25-28, 7:3-8, 7:14-8:20);

in response to detecting an attempt to access the artificial memory region, determining an export table entry in the export table of the dynamically-linked library that is associated with the attempt to access the artificial memory region (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:10-12, 3:13-19, 6:25-7:8, 7:12-20, 7:27-8:20); and

determines based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral (*see, e.g.,* FIGURES 2 and 4; Spec. at 3:10-12, 3:20-4:4, 6:16-18, 6:29-7:8, 7:20-26, 8:2-4, 8:7-20).

Claim 14 recites:

An apparatus for detecting in a subject file viral code that uses calls to an operating system, comprising:

a processor (*see, e.g.,* FIGURES 1 and 2, Spec. at 5:9-23);

a memory (*see, e.g.,* FIGURES 1 and 2, Spec. at 5:9-23);

a CPU emulator (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:2-5, 3:13-15, 3:21-25, 6:4-6, 6:18-24, 7:20-26);

a memory manager component that creates an artificial memory region spanning at least a portion of the memory associated with an export table of a dynamically-linked library (*see, e.g.,* FIGURES 2, 3, and 4;-Spec. at 3:6-8, 3:17-19, 3:22-24, 4:2-4; 5:30-6:7, 6:13-18, 7:14-19); and

a monitor component, wherein the CPU emulator emulates execution of at least a portion of computer executable code in the subject file (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:2-5, 3:13-15, 3:21-25, 6:4-6, 6:18-24, 7:20-26), and the monitor component:

monitors attempts by the emulated computer executable code to access the artificial memory region (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:10-12, 3:13-18, 3:21-22, 3:25-26, 6:6-12, 6:19-20, 6:25-28, 7:3-8, 7:14-8:20);

in response to detecting an attempt to access the artificial memory region, determining an export table entry in the export table of the dynamically-linked library that is associated with the attempt to access the artificial memory region (*see, e.g.,* FIGURES 2, 3, and 4; Spec. at 3:10-12, 3:13-19, 6:25-7:8, 7:12-20, 7:27-8:20); and

determines based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral (*see, e.g.,* FIGURES 2 and 4; Spec. at 3:10-12, 3:20-4:4, 6:16-18, 6:29-7:8, 7:20-26, 8:2-4, 8:7-20).

GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

1. Are Claims 1, 10, 11, 12, and 14 allowable under 35 U.S.C. § 103(a) over U.S. Patent No. 6,192,512 issued to Chess ("*Chess*") in view of U.S. Patent No. 5,851,057 issued to Nachenberg ("*Nachenberg*")?

2. Is Claim 21 allowable under 35 U.S.C. § 103(a) over 35 U.S.C. 103(a) as being unpatentable over *Chess* and *Nachenberg* in view of U.S. Patent No. 5,398,196 to Chambers ("*Chambers*")?

ARGUMENT

For at least the following reasons, Appellants respectfully submit that the Examiner's rejections of Claims 1, 4, 8-16 and 20-23 are improper and should be reversed by the Board.

I. The Legal Standard for Obviousness

The question raised under 35 U.S.C. § 103 is whether the prior art taken as a whole would suggest the claimed invention taken as a whole to one of ordinary skill in the art at the time of the invention. One of the three basic criteria that must be established by an Examiner to establish a *prima facie* case of obviousness is that "the prior art reference (or references when combined) must teach or suggest ***all the claim limitations***." *See* M.P.E.P. § 706.02(j) citing *In re Vaeck*, 947 F.2d 488, 20 U.S.P.Q.2d 1438 (Fed. Cir. 1991) (emphasis added). "***All words*** in a claim must be considered in judging the patentability of that claim against the prior art." *See* M.P.E.P. § 2143.03 citing *In re Wilson*, 424 F.2d 1382, 1385 165 U.S.P.Q. 494, 496 (C.C.P.A. 1970) (emphasis added).

In addition, even if all elements of a claim are disclosed in various prior art references, which is certainly not the case here as discussed below, the claimed invention taken as a whole still cannot be said to be obvious without some reason why one of ordinary skill at the time of the invention would have been prompted to modify the teachings of a reference or combine the teachings of multiple references to arrive at the claimed invention.

II. Issue 1 - Claims 1, 4, 8-16, 20, 22, and 23 are Allowable under 35 U.S.C. § 103(a) over the Proposed *Chess-Nachenberg* Combination

The Examiner rejects Claims 1, 4, 8-16, 20, 22, and 23 under 35 U.S.C. § 103(a) as being unpatentable over the proposed *Chess-Nachenberg* combination. Appellants respectfully submit that these rejections are improper and should be reversed by the Board.

A. The Proposed *Chess-Nachenberg* Combination Fails to Disclose Various Claim Limitations

At a minimum, the proposed *Chess-Nachenberg* combination fails to disclose, teach, or suggest the following limitations recited in Claim 47, which Appellants discuss as an example:

- in response to detecting an attempt to access the artificial memory region, determining an export table entry in the export table of the dynamically-

linked library that is associated with the attempt to access the artificial memory region; and

- determining based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral

1. "*in response to detecting an attempt to access the artificial memory region, determining an export table entry in the export table of the dynamically-linked library that is associated with the attempt to access the artificial memory region*"

For example, the *Chess-Nachenberg* combination fails to teach, suggest, or disclose "in response to detecting an attempt to access the artificial memory region, determining an export table entry in the export table of the dynamically-linked library that is associated with the attempt to access the artificial memory region" as recited by Claim 1.

The Examiner rejected Claim 1 in a Non-Final Office Action issued October 22, 2008 (the "Non-Final Office Action") and maintained this rejection in the Final Office Action. However, the rejection, as presented in the Non-Final Office Action and the main body of the Final Office Action, failed to address the actual language of this element. The rejection improperly paraphrased the wording of this element in comparing the alleged teachings of both *Chess* and *Nachenberg* to the claimed subject matter. Specifically, the Office Action asserted that "*Chess* discloses . . . in response to detecting an attempt to access the artificial memory region, ***determining a source program that is associated with the attempt to access*** the artificial memory region." Non-Final Office Action at pp. 2-3; Final Office Action at pp. 2-3 (emphasis added). While Appellants do not agree with the Examiner's reading of *Chess* (as discussed further below), Appellants noted in their response to the Non-Final Office Action that this description erroneously paraphrased the language of Claim 1. In particular, Claim 1 recites "in response to detecting an attempt to access the artificial memory region, ***determining an export table entry*** in the export table of the dynamically-linked library ***that is associated with the attempt to access the artificial memory region***" (emphasis added).

Moreover, the Non-Final Office Action simply asserted that *Nachenberg* teaches "monitoring these entry points . . . to ***determine whether a virus is present***." Non-Final Office Action, p. 4, emphasis added. While Appellants did not necessarily agree with the Non-Final Office Action's reading of *Nachenberg* either, Appellants noted in their response to the Non-Final Office Action that this description also erroneously paraphrased the language of Claim 1. Again, Claim 1 recites "in response to detecting an attempt to access

the artificial memory region, ***determining an export table entry*** in the export table of the dynamically-linked library ***that is associated with the attempt to access the artificial memory region***" (emphasis added).

The rejection of Claim 1 presented in the body of the Final Office Action did not change the misquoting of the language of Claim 1. *See* Final Office Action at pp. 2-3. The "Response to Arguments" section of the Final Office Action, however, attempted to overcome this deficiency of the rejection by providing a different description of the cited references. *See* Final Office Action at p. 6-7. In doing so, however, the Examiner again incorrectly paraphrased the language of Claim 1. Specifically, the Examiner asserted simply that the system of *Nachenberg* "report[] which ***entry point*** is infected" (emphasis added). Thus, in attempting to correct the deficiencies of the previous rejection, the Examiner's revised argument improperly substituted the "an export table entry in the export table" language recited by Claim 1 with the phrase "an entry point [of the virus]." *Cf.* Final Office Action at p.3 (referring to "an entry point ***for viruses***") (emphasis added). Thus, neither the rejection as presented in the Non-Final Office Action (and the main body of the Final Office Action) nor as presented in the "Response to Arguments" section of the Final Office Action addresses the actual language of this element of Claim 1. Appellants respectfully note that "All words in a claim must be considered in judging the patentability of that claim against the prior art." *In re Wilson*, 424 F.2d 1382, 1385; 165 U.S.P.Q. 494, 496 (C.C.P.A. 1970)); *see also* M.P.E.P. § 2143.03. Thus, because the rejection fails to address the actual words of Claim 1 itself, the rejection is improper.

Furthermore, the Examiner's descriptions of the cited references mischaracterized several aspects of the systems described by *Chess* and *Nachenberg*. For example, the Final Office Action alleged that "*Nachenberg* teaches monitoring entry points of viruses by emulating the applications, determining where virtual memory has been modified and reporting which entry point is infected." Final Office Action at p. 6. Appellants respectfully note that this assertion inaccurately summarizes the cited portion of *Nachenberg*. Instead, the cited portion states, *inter alia*, that:

> The VDS (400) uses the scanning module (424) to scan pages of the virtual memory (434) that were either modified or emulated through for signatures of polymorphic viruses and uses stochastic information obtained during the emulation, such as instruction usage profiles, to detect metamorphic viruses. ***If the scanning module (424) or VDS (400) detects a virus, the VDS reports that the file (100) is infected.***

Nachenberg at col. 4, ll. 59-64, emphasis added.

The cited portion does not indicate that the system of *Nachenberg* "report[s] ***which entry point*** is infected" (emphasis added), as the Examiner contends, but only that a virus has been detected and/or that the file (100) is infected. Thus, the rejection of Claim 1 presented in the Non-Final Office Action fails to address the language of Claim 1 itself, and the rejection of Claim 1 presented in the Final Office Action plainly mischaracterizes the *Nachenberg* reference.

Additionally, the rejection of Claim 1 also mischaracterized *Chess*. According to both the Non-Final Office Action and the Final Office Action, *Chess* allegedly "discloses . . . in response to detecting an attempt to access the artificial memory region, determining a source program that is associated with the attempt to access the artificial memory region." *See, e.g.,* Final Office Action at p. 2. However, the cited portion of *Chess* merely discloses:

> By example, if it is found that the source program unexpectedly attempts to access a virtualized mass storage medium, and/or to create one or more copies of itself in a region of virtualized memory, or in a virtual file, then the external program can be informed of the potentially viral nature ***of the source program***.
>
> *Chess* at col. 4, ll. 49-54, emphasis added.

Therefore, the cited portion of *Chess* indicates only that "if it is found that the source program unexpectedly attempts to access a virtualized mass storage medium . . . ***then the external program can be informed*** of the potentially viral nature of the source program." *Id.*, emphasis added. *Chess* does not disclose "in response to detecting an attempt to access the artificial memory region, ***determining a source program that is associated with the attempt*** to access the artificial memory region." Indeed, the cited portion refers to only "*the* source program," and not to multiple source programs between which the system determines. Consequently, the Final Office Action's characterization of *Chess* as disclosing "determining a source program," or for that matter determining any element, that is "associated with the attempt to access the artificial memory region" in response to detecting an attempt to access the artificial memory region was plainly inaccurate. Consequently, *Chess* does not disclose, "in response to detecting an attempt to access the artificial memory region," determining any element "that is associated with the attempt to access the artificial memory region."

Thus, for at least these reasons, the proposed *Chess-Nachenberg* combination fails to disclose "in response to detecting an attempt to access the artificial memory region,

determining an export table entry in the export table of the dynamically-linked library that is associated with the attempt to access the artificial memory region" as recited by Claim 1.

2. "*determining based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral*"

As an additional distinction, the proposed *Chess-Nachenberg* combination fails to teach, suggest, or disclose "determining based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral" as recited by Claim 1. In addressing this element, the Non-Final Office Action erroneously paraphrased the language of Claim 1 once again. Specifically, the Non-Final Office Action asserted simply that "*Chess* discloses . . . determining *based on the attempt to access the artificial memory region* that the emulated computer executable code is viral." Office Action, pp. 3-4, emphasis added. Appellants noted in their response to the Non-Final Office Action that this description again erroneously paraphrased the language of Claim 1. In particular, Claim 1 recites "determining *based on the export table entry associated with the attempt to access* the artificial memory region that the emulated computer executable code is viral" (emphasis added). Thus, the Office Action fails to cite to any portion of *Chess* or *Nachenberg* that allegedly discloses "determining based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral" as recited by Claim 1, and the proposed *Chess-Nachenberg* combination fails to disclose this element.

The rejection of Claim 1 presented in the main body of the Final Office Action did not change the misquoting of the language of Claim 1. *See* Final Office Action at p. 2. The "Response to Arguments" section of the Final Office Action, however, attempted to overcome this deficiency of the rejection by providing a different description of the cited references. *See* Final Office Action at p. 7. In doing so, however, the Examiner again mischaracterizes *Nachenberg*. The Examiner cites to a portion of *Nachenberg* (i.e., *Nachenberg* at col. 6, ll. 53-64) describing "a flow chart illustrating steps performed by a typical virus when infecting the host file 100." *See Nachenberg* at col. 6, ll. 30-31. Thus, this portion of *Nachenberg* cited by the Examiner describes how a typical virus may operate. Contrary to the Examiner's assertions, however, the cited portion of *Nachenberg* describes *detecting* viruses based on other characteristics. Specifically, *Nachenberg* staets instead that:

> [T]he VDS (400) uses the scanning module (424) to ***scan pages*** of the virtual memory (434) that were either modified or emulated through ***for signatures of polymorphic viruses and uses stochastic information*** obtained during the emulation, such as instruction usage profiles, ***to detect metamorphic viruses***. If the scanning module (424) or VDS (400) detects a virus, the VDS reports ***that the file (100) is infected.***"
>
> *Nachenberg* at col. 4, ll. 59-64, emphasis added.

The mere fact *Nachenberg* teaches that "export table 122" is among the parts of "file (100)" a virus can affect, does not mean that *Nachenberg* discloses "determining ***based on the export table entry associated with the attempt to access*** the artificial memory region that the emulated computer executable code is viral" (emphasis added), as *Nachenberg* clearly discloses another way of detecting viruses -- that is, by scanning pages of virtual memory for polymorphic viruses and by using stochastic information to detect metamorphic viruses. The technique described by *Nachenberg* does not depend in any manner on which export table entry is associated with the attempt to access the artificial memory region. Thus, *Nachenberg* does not disclose "determining ***based on the export table entry associated with the attempt to access*** the artificial memory region that the emulated computer executable code is viral" (emphasis added).

Thus, for at least these reasons, the proposed *Chess-Nachenberg* combination fails to disclose "determining based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral" as recited by Claim 1.

B. Conclusions

Therefore, the proposed *Chess-Nachenberg* combination fails to disclose, teach, or suggest at least the above-discussed limitations recited in Claim 1. For at least these reasons, the Examiner has not established a *prima facie* case of obviousness. Accordingly, Appellants respectfully request that the Board reverse the rejections of independent Claim 1 and its dependent claims.

Although of differing scope from independent Claim 1, independent Claims 10-12 and 14 each recite certain limitations that, for reasons substantially similar to those discussed with reference to independent Claim 1, are not disclosed, taught, or suggested by the references, alone or in combination, as set forth by the Examiner. For at least these reasons,

Appellants respectfully request that the Board reverse the rejections of independent Claims 10-12 and 14 and their respective dependent claims.

III. Issue 2 - Claim 21 is Allowable under 35 U.S.C. § 103(a) over the Proposed *Chess-Nachenberg-Chambers* Combination

The Examiner rejects Claim 21 under 35 U.S.C. 103(a) as being unpatentable over the modified Chess and Nachenberg system as applied to claim 1 above, and further in view of U.S. Patent No. 5,398,196 to Chambers ("*Chambers*").

Claims 21 depends from independent Claim 1, which has been shown above to be allowable. The Examiner does not allege that *Chambers* makes up for the above-discussed deficiencies of the proposed *Chess-Nachenberg* combination. Accordingly, dependent Claim 21 is allowable over the cited references at least for depending from an allowable independent claim.

For at least these reasons, Appellants respectfully submit that these rejections of Claims 1, 4, 8-16 and 20-23 are improper and respectfully request the Board to reverse these rejections.

CONCLUSION

Appellants have demonstrated that the present invention, as claimed, is clearly distinguishable over the prior art cited by the Examiner. Therefore, Appellants respectfully requests the Board to reverse the final rejections and instruct the Examiner to issue a Notice of Allowance with respect to all pending claims.

The Commissioner is hereby authorized to charge $540.00 for filing this Brief in support of an Appeal and the $130.00 one-month extension of time fee to Deposit Account No. 02-0384 of Baker Botts, L.L.P. and the $130.00 fee for a one-month extension of time No other fees are believed due; however, the Commissioner is authorized to charge any additional fees or credits to Deposit Account No. 02-0384 of Baker Botts, L.L.P.

Respectfully submitted,

BAKER BOTTS L.L.P.
Attorneys for Appellants

Todd A. Cason
Reg. No. 54,020
(214) 953-6542

Dated: November 4, 2009

Correspondence Address:

at Customer No. **05073**

APPENDIX A

Pending Claims

1. A method of detecting viral code in subject files, comprising:

creating an artificial memory region spanning one or more components of the operating system, wherein the artificial memory region is associated with an export table of a dynamically-linked library;

emulating execution of at least a portion of computer executable code in a subject file;

monitoring attempts by the emulated computer executable code to access the artificial memory region;

in response to detecting an attempt to access the artificial memory region, determining an export table entry in the export table of the dynamically-linked library that is associated with the attempt to access the artificial memory region; and

determining based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral.

4. The method of claim 1, further comprising:

emulating functionality of an identified operating system call while monitoring the operating system call to determine whether the computer executable code is viral.

8. The method of claim 1, further comprising monitoring access by the emulated computer executable code to dynamically linked functions.

9. The method of claim 8, wherein the artificial memory region spans a jump table containing pointers to the dynamically linked functions.

10. A program storage device readable by a machine, tangibly embodying a program of instructions executable by the machine to perform method steps for detecting viral code in subject files, the method steps comprising:

creating an artificial memory region spanning one or more components of the operating system, wherein the artificial memory region is associated with an export table of a dynamically-linked library;

emulating execution of at least a portion of computer executable code in a subject file;

monitoring attempts by the emulated computer executable code to access the artificial memory region;

in response to detecting an attempt to access the artificial memory region, determining an export table entry in the export table of the dynamically-linked library that is associated with the attempt to access the artificial memory region; and

determining based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral.

11. A computer system, comprising:

a processor; and

a program storage device readable by the computer systems, tangibly embodying a program of instructions executable by the processor to perform method steps for detecting viral code in subject files, the method comprising:

creating an artificial memory region spanning one or more components of the operating system, wherein the artificial memory region is associated with an export table of a dynamically-linked library;

emulating execution of at least a portion of computer executable code in a subject file;

monitoring attempts by the emulated computer executable code to access the artificial memory region; and

determining based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral.

12. A computer data signal embodied in a computer-readable medium which embodies instructions executable by a computer for detecting in a subject file viral code that uses calls to an operating system, the signal comprising:

a first segment comprising CPU emulator code, wherein the CPU emulator code emulates execution of at least a portion of computer executable code in the subject file;

a second segment comprising memory manager code, wherein the memory manager code creates an artificial memory region spanning components of the operating system, wherein the artificial memory region is associated with an export table of a dynamically-linked library; and

a third segment comprising monitor code, wherein the monitor code:

monitors attempts by the emulated computer executable code to access the artificial memory region;

in response to detecting an attempt to access the artificial memory region, determining an export table entry in the export table of the dynamically-linked library that is associated with the attempt to access the artificial memory region; and

determines based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral.

13. The computer data signal of claim 12, further comprising:

a fourth segment comprising analyzer code, wherein the analyzer code emulates functionality of the identified operating system call to determine whether the computer executable code is viral.

14. An apparatus for detecting in a subject file viral code that uses calls to an operating system, comprising:

a processor;

a memory;

a CPU emulator;

a memory manager component that creates an artificial memory region spanning at least a portion of the memory associated with an export table of a dynamically-linked library; and

a monitor component, wherein the CPU emulator emulates execution of at least a portion of computer executable code in the subject file, and the monitor component:

monitors attempts by the emulated computer executable code to access the artificial memory region;

in response to detecting an attempt to access the artificial memory region, determining an export table entry in the export table of the dynamically-linked library that is associated with the attempt to access the artificial memory region; and

determines based on the export table entry associated with the attempt to access the artificial memory region that the emulated computer executable code is viral.

15. The apparatus of claim 14, further comprising:

an auxiliary component; and

an analyzer component,

wherein the auxiliary component emulates functionalities of an identified operating system call, and the monitor component monitors the operating system call to determine whether the computer executable code is viral, while emulation continues.

16. The apparatus of claim 15, wherein the auxiliary component emulates functionalities of the operating system call.

20. The apparatus of claim 14, wherein the artificial memory region created by the memory manager component spans a jump table containing pointers to dynamically linked functions, and the monitor component monitors access by the emulated computer executable code to the dynamically linked functions.

21. The method of claim 1, further comprising:

monitoring accesses by the emulated computer executable code to the artificial memory region to detect looping; and

determining based on a detection of looping that the emulated computer executable code is viral.

22. The method of claim 1, wherein creating an artificial memory region comprises creating a custom version of the export table with predetermined values for the entry points.

23. The method of claim 1, further comprising:

monitoring access by the emulated computer executable code to dynamically linked functions; and

determining based on attempted access to dynamically linked functions that the emulated computer executable code is viral.

APPENDIX B

Evidence Appendix

Other than the references cited during prosecution, no evidence was submitted pursuant to 37 C.F.R. §§ 1.130, 1.131, or 1.132, and no other evidence was entered by the Examiner and relied upon by Appellant in the Appeal.

US006192512B1

(12) **United States Patent**
Chess

(10) **Patent No.: US 6,192,512 B1**
(45) **Date of Patent: Feb. 20, 2001**

(54) **INTERPRETER WITH VIRTUALIZED INTERFACE**

(75) Inventor: **David M Chess**, Mohegan Lake, NY (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Under 35 U.S.C. 154(b), the term of this patent shall be extended for 0 days.

(21) Appl. No.: **09/160,117**

(22) Filed: **Sep. 24, 1998**

(51) **Int. Cl.**[7] **G06F 9/45**; G06F 11/30

(52) **U.S. Cl.** **717/5**; 717/4; 714/38; 709/328; 713/200

(58) **Field of Search** 395/704, 705; 717/5, 4; 714/37; 709/328; 713/200

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,440,723		8/1995	Arnold et al.	714/2
5,452,442		9/1995	Kephart	714/38
5,475,843	*	12/1995	Halviatti et al.	717/4
5,485,575		1/1996	Chess et al.	714/38
5,526,523	*	6/1996	Straub et al.	709/328
5,572,590		11/1996	Chess	713/200
5,572,675	*	11/1996	Bergler	709/328
5,602,982	*	2/1997	Judd et al.	345/326
5,613,002		3/1997	Kephart et al.	713/200
5,636,371	*	6/1997	Yu	703/26
5,706,453	*	1/1998	Cheng et al.	345/347
5,708,774	*	1/1998	Boden	714/38

(List continued on next page.)

OTHER PUBLICATIONS

IBM Corporation; "Implementation of Common User Access Controls under AIX Motif Environment". IBM Technical Disclosure Bulletin, vol. 36, iss 3, pp. 163–164, Mar. 1993.*

Ito et al., "A hardware/Software Co–stimulation Environment for Micro–processor Design with HDL Simulator and OS interface". IEEE/IEE Electronic Library[online], Proceedings of the ASP–DAC '97 Asia and South Pacific Design Automation Conference, Jan. 1997.*

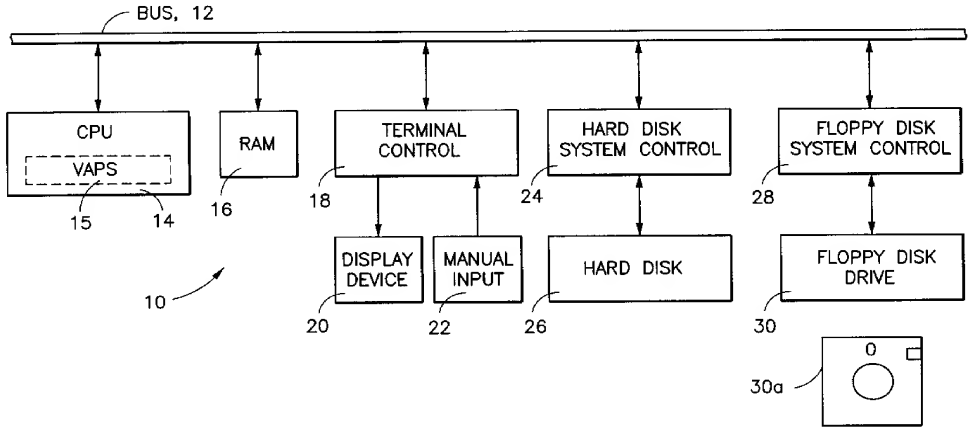
Fleet et al., "Automated Validation of Operational Flight Programs (OFPs) and Flight Training Simulators". IEEE/IEE Electronic Library[online], Proceedings of the IEEE'94 National Aerospace and Electronics Conference, Jan. 1997.*

Primary Examiner—Mark R. Powell
Assistant Examiner—Kelvin E. Booker
(74) *Attorney, Agent, or Firm*—Ohlandt, Greeley, Ruggiero & Perle, L.L.P.; David M. Shofi, Esq.; IBM Corporation

(57) **ABSTRACT**

A computer application program subsystem **(100)** includes a program interpreter **(120)** and an application program interface **(API 110)** through which an external program requests an execution of a program of interest, such as a macro, in a specified simulated environment. The external program that requests the execution of the program of interest may further specify a simulated application state. The program of interest is written in a program language that the interpreter can interpret. The subsystem further includes an output path for returning to the external program at least one indication of what action or actions the program of interest would have taken if the program of interest had been run in a real environment that corresponds to the specified simulated environment. The output path may be implemented using a callback function that is triggered upon the occurrence of an instruction of the program of interest satisfying at least one notification criterion, and/or upon the occurrence of the program of interest satisfying at least one termination criterion. The methods and apparatus can be useful in detecting an occurrence of viral behavior in a macro by interpreting the macro in the specified virtual environment and virtual application state, and then notifying the external program when the macro performs some predetermined activity, such as writing data to some predetermined region of system memory.

20 Claims, 3 Drawing Sheets



U.S. PATENT DOCUMENTS

5,734,865	*	3/1998	Yu	709/250
5,734,907	*	3/1998	Jarossay et al.	717/8
5,754,760	*	5/1998	Warfield	714/38
5,790,117	*	8/1998	Halviatti et al.	345/333
5,835,089	*	11/1998	Skarbo et al.	345/335
5,860,010	*	1/1999	Attal	717/6
5,920,725	*	7/1999	Ma et al.	717/11
5,922,054	*	7/1999	Bibayan	709/328
5,961,582	*	10/1999	Gaines	709/328
5,974,256	*	10/1999	Matthews et al.	717/5
6,026,238	*	2/2000	Bond et al.	717/5
6,067,639	*	5/2000	Rodrigues et al.	714/38
6,096,095	*	8/2000	Halstead	717/5
6,101,607	*	8/2000	Bachand et al.	709/328
6,108,799	*	8/2000	Boulay et al.	714/38

* cited by examiner

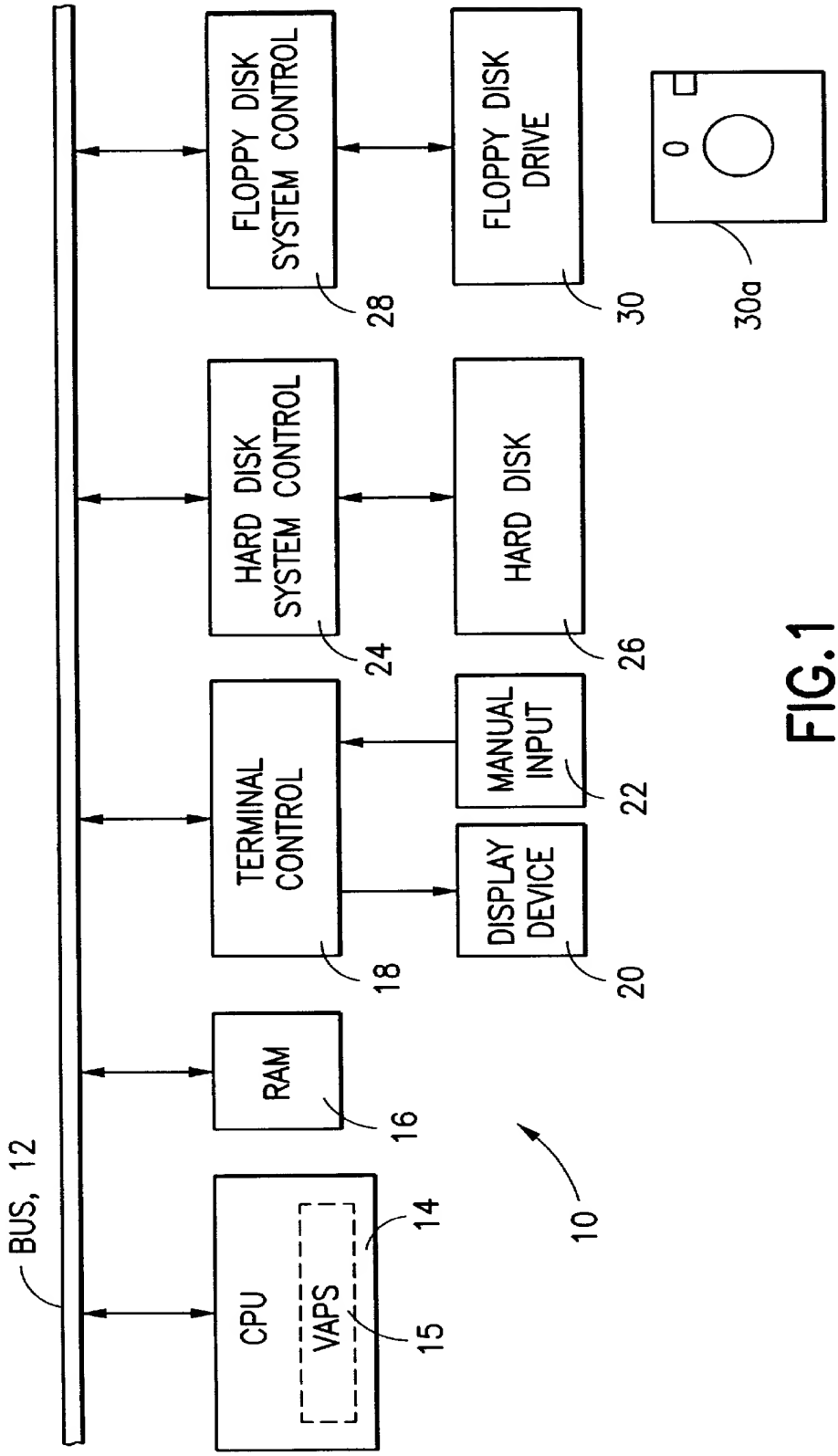

BUS, 12
CPU
VAPS
14
15
RAM
16
TERMINAL CONTROL
18
DISPLAY DEVICE
20
MANUAL INPUT
22
HARD DISK SYSTEM CONTROL
24
HARD DISK
26
FLOPPY DISK SYSTEM CONTROL
28
FLOPPY DISK DRIVE
30
30a
10

FIG. 1

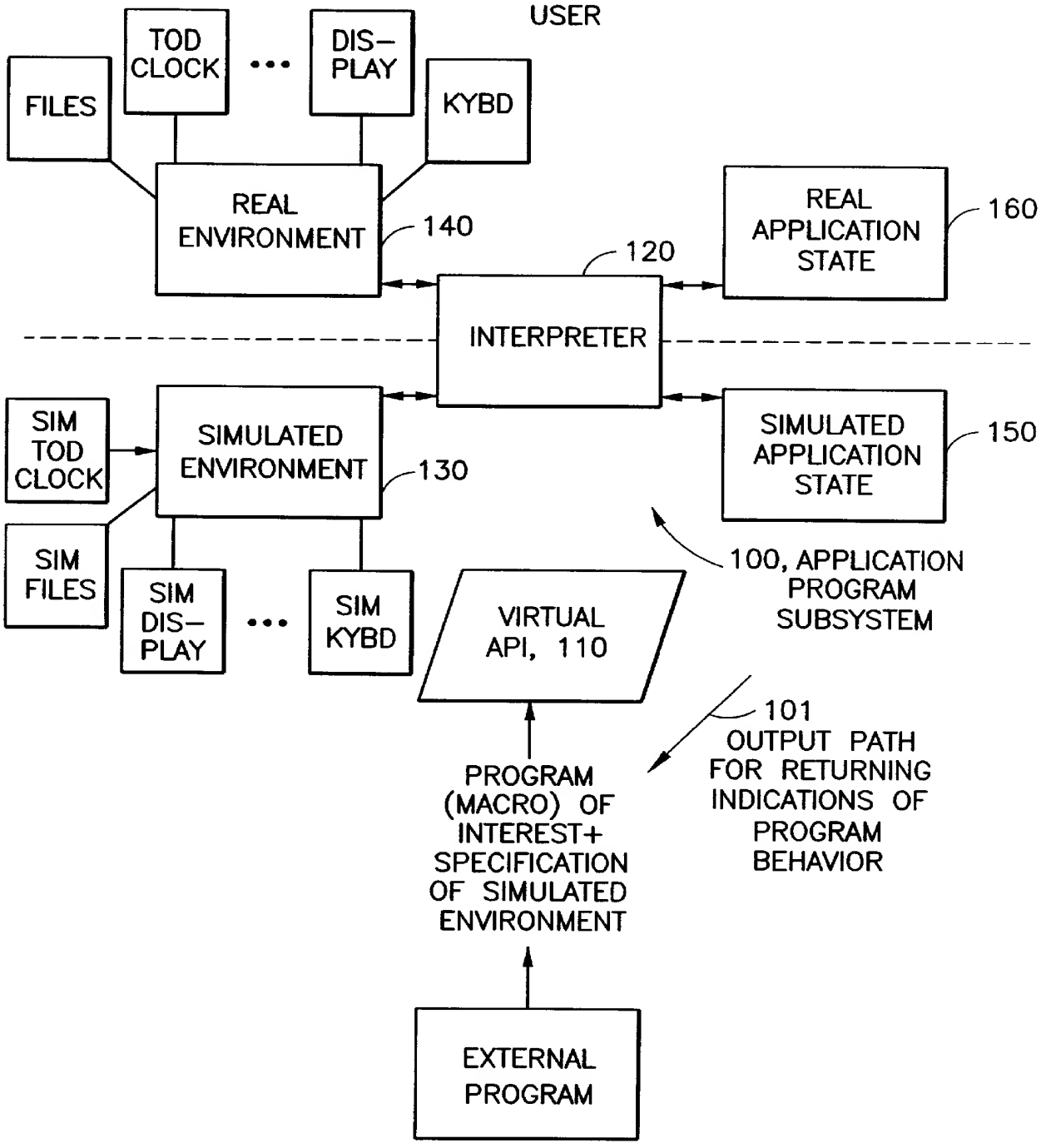

USER
TOD CLOCK
DIS-PLAY
FILES
KYBD
REAL ENVIRONMENT
140
120
REAL APPLICATION STATE
160
INTERPRETER
SIM TOD CLOCK
SIMULATED ENVIRONMENT
130
SIMULATED APPLICATION STATE
150
SIM FILES
SIM DIS-PLAY
SIM KYBD
VIRTUAL API, 110
100, APPLICATION PROGRAM SUBSYSTEM
101
OUTPUT PATH FOR RETURNING INDICATIONS OF PROGRAM BEHAVIOR
PROGRAM (MACRO) OF INTEREST+ SPECIFICATION OF SIMULATED ENVIRONMENT
EXTERNAL PROGRAM

FIG.2

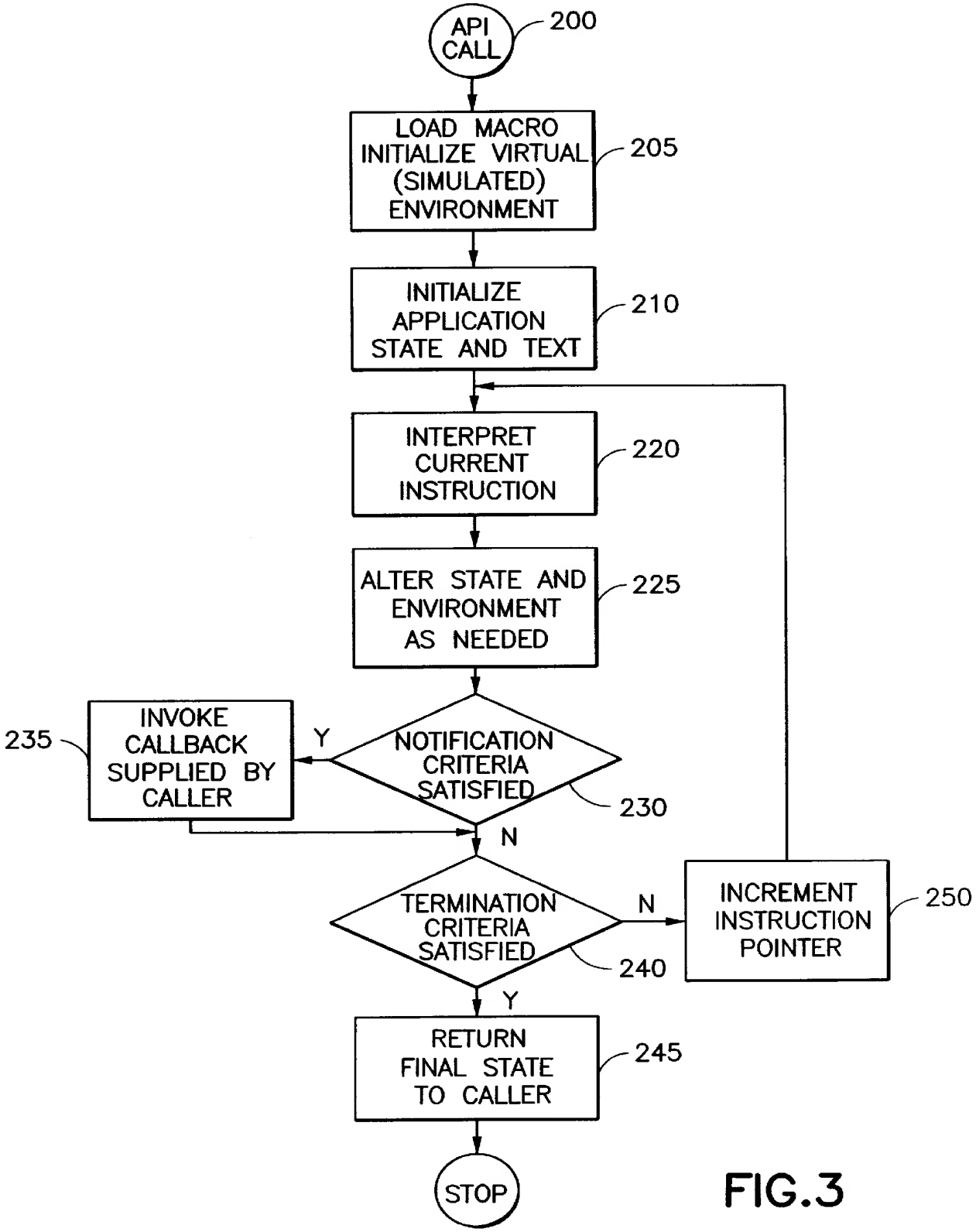

API CALL
200
LOAD MACRO INITIALIZE VIRTUAL (SIMULATED) ENVIRONMENT
205
INITIALIZE APPLICATION STATE AND TEXT
210
INTERPRET CURRENT INSTRUCTION
220
ALTER STATE AND ENVIRONMENT AS NEEDED
225
NOTIFICATION CRITERIA SATISFIED
230
Y
INVOKE CALLBACK SUPPLIED BY CALLER
235
N
TERMINATION CRITERIA SATISFIED
240
N
INCREMENT INSTRUCTION POINTER
250
Y
RETURN FINAL STATE TO CALLER
245
STOP

FIG.3

INTERPRETER WITH VIRTUALIZED INTERFACE

FIELD OF THE INVENTION

This invention is generally directed to computer software and, in particular, to a class of computer programs known as interpreters.

BACKGROUND OF THE INVENTION

As data processing systems become more powerful, more complex, and more highly interconnected, interpreted languages of various kinds are growing in importance. In non-interpreted languages, source programs are converted into the machine language for some particular CPU, and then stored, transmitted and executed in machine-language form. Computer programs that perform this function are typically referred to as compilers.

However, when using a computer program known as an interpreter the source programs are stored, transmitted and executed in a higher-level language. The interpreter is required at execution time to read the source program and carry out the instructions that it contains.

Interpreted languages tend to execute slower than other types of languages. That is, an interpreted source program may execute more slowly than the same source program that has been previously compiled into the machine code of the host CPU. However, the use of interpreters provides several advantages, including an ability to execute the same program on many different CPUs and operating systems (as long as each CPU and operating system has an interpreter for the language).

Interpreters are often embedded inside application program subsystems. For example, one widely used word processing program contains an interpreter for a specific language in which the word processor macros are written. The LotusNotes™ product contains an interpreter for LotusScript™, one of the languages in which LotusNotes™ macros are written. Also, several Web browsers contain interpreters for Java™, a language in which programs of various kinds are written, and made available on the World Wide Web (WWW). In many cases, these application program subsystems include application programming interfaces (APIs) which allow other programs to interact with the application program subsystem. One common feature offered by many APIs is the ability for an external program to access, create and execute programs written in the application program subsystem's interpreted language.

There are various types of programs, including anti-virus programs and general security programs, which may need to examine interpreted programs designed for application program subsystems. For a variety of purposes, such an external program may need to determine at least a subset of the actions that a given interpreted-language program would take, if it were to be executed by the interpreter contained in some application subsystem. Since existing APIs do not provide powerful features to allow this determination, designers of these external programs are faced with the choice of either implementing at least a subset of the interpreters themselves, or not supporting detailed examination of programs written for these interpreters. The former course of action is difficult, as implementing an interpreter for a powerful macro language can be costly and time-consuming, and it is not easy to ensure that the implemented interpreter actually behaves as the real one would in all relevant cases. The latter course of action is dangerous, as it may lead to exposure to viruses or security attacks.

SUMMARY OF THE INVENTION

This invention is directed to software interpreters. Specifically, the teachings of this invention are directed to providing a software interpreter within an application subsystem, and includes a virtualized interface to external programs for the evaluation of their behavior within the subsystem.

For security, virus detection and other purposes, external programs may need to determine what a given program would do if executed under a given interpreter. With the proliferation of complex interpreted languages, it may not be feasible for each of these external programs to contain its own implementations of each interpreter. On the other hand, application programming interfaces (APIs) known in the art only allow external programs to request the interpreter to execute a program, and not to determine what the program would do it were to be executed. The present invention solves this problem by providing an interpreter with an API allowing access to a virtualized mode of the interpreter.

In a first aspect this invention provides a computer subsystem having an interpreter that supports one or more programming languages. The computer subsystem includes an application programming interface (API) through which an external program may request simulated execution of a given program, written in one of the supported languages, in a specific simulated environment. The computer subsystem may thus simulate execution of the given program that is passed to it, and return to the external program one or more indications of what actions the given program would have taken had it actually run in the application program subsystem environment.

A method is also provided for predicting the actions of a given program within an application program subsystem having an interpreter. The method includes steps of: (a) receiving an API call for prediction; (b) initializing a virtual environment based on a real environment; (c) initializing application state information; (d) interpreting instructions of the given program; and (e) altering the application state information and the virtual environment in response to the interpreting step.

A method is disclosed for exercising a macro with an application program subsystem, having a macro interpreter, so as to detect a presence of potential viral activity. The method includes the steps of: (a) making an API call with a program, the API call identifying the macro and specifying an initial virtual environment within which the macro is to be interpreted; (b) interpreting in turn individual instructions of the macro; (c) altering the virtual environment in response to interpreted instructions; and (d) notifying the program upon the occurrence of an alteration to the virtual environment that triggers a predetermined notification criterion. The step of notifying the program can also take place upon the occurrence of the interpretation of the macro triggering a predetermined termination criterion. The API call may further specify an initial virtual application state within which the macro is to be interpreted.

In accordance with a further aspect of this invention there is provided a system and a method for executing a macro with a data processing system. The method includes a first step of generating an application program interface (API) call for interpreting a macro of interest in a virtual environment so as to execute all or a part of the macro of interest in the virtual environment. The step of generating includes a step of defining the virtual environment, wherein the virtual environment is defined at least in part through the use of at least one callback function that is invoked upon an

occurrence of a virtual execution of the macro satisfying a predetermined callback function invocation criterion. The step of generating further includes a step of initializing state information. A next step of the method interprets in turn individual instructions of the macro, and alters at least one of the state information and the virtual environment in response to interpreted instructions.

The predetermined callback function invocation criterion can be, by example, an interpreted macro instruction attempting to perform file I/O, or attempting to access memory, or attempting to access a predetermined portion of a mass memory device.

The virtual environment includes simulated physical and logical devices that are either all present as real physical and logical devices in the data processing system, or that are not all present as real physical and logical devices in the data processing system.

BRIEF DESCRIPTION OF THE DRAWING

Further objects, features, and advantages of the present invention will become apparent from a consideration of the following detailed description of the invention when read in conjunction with the attached drawing figures, wherein:

FIG. **1** is a circuit block diagram of the hardware environment to which the present invention can be applied;

FIG. **2** is a conceptual block diagram of the software environment of the present invention; and

FIG. **3** is a logic flow diagram of an embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

The present invention includes an improved application subsystem that provides an API that includes a "virtualized" interface to an interpreter. The virtualized interface allows external programs to determine what a given program in the interpreted language would do, were it to be run on the interpreter in a certain environment. By providing the API, an application subsystem can enable external programs to more easily examine programs written for its interpreter, and thus allow enhanced virus protection and security programs to be written to support its interpreter.

One important aspect of the teaching of this invention is an interface whereby an external program can describe to the subsystem at least: (1) a program written for the subsystem's interpreter (the "macro") to be executed (or partially executed) in a virtual mode, (2) a state of the external environment to be simulated during virtual execution of the macro, and (3) a set of conditions under which virtual execution of the macro should halt.

The macro itself will typically be provided in the form of the text of the program, either in a buffer, in a file whose name is passed via the API, or in some other form that will enable the subsystem to find and load the macro. Parts of the description of the environment may be in the form of callback routines, provided by the external program, which the subsystem will invoke whenever the virtual execution of the macro requires some data from the outside world, or when the virtual execution of the macro takes some action that would effect the outside world. Other parts of the description of the environment may be in the form of normal binary data.

As employed herein a callback can be implemented by passing a pointer to a function that will execute upon the occurrence of some event or condition. As but one example, the API can pass the interpreter a pointer to a routine that is executed whenever the virtual interpretation of the macro results in an attempt to perform file I/O.

FIG. **1** is a block diagram of a system **10** that is suitable for practicing the teaching of the present invention. A bus **12** is comprised of a plurality of signal lines for conveying addresses, data and controls between a central processing unit (CPU) **14** and a number of other system bus units. A random access memory (RAM) **16** is coupled to the system bus **12** and provides program instruction storage and working memory for the CPU **14**. A terminal control subsystem **18** is coupled to the system bus **14** and provides outputs to a display device **20**, typically a CRT monitor, and receives inputs from a manual input device **22**, such as a keyboard or pointing device. A hard disk control subsystem **24** bidirectionally couples a rotating fixed storage medium, such as a hard disk **26**, to the system bus **12**. The control **24** and hard disk **26** provide mass storage for CPU instructions and data. A removable medium control subsystem, such as a floppy disk system control **28** along with a floppy disk drive **30**, is useful as an input means in the transfer of computer files from a floppy diskette **30**a to system memory via the system bus **12**.

The components illustrated in FIG. **1** may be embodied within a personal computer, a portable computer, a workstation, a minicomputer or a supercomputer. As such, the details of the physical embodiment of the data processing system **10**, such as the structure of the bus **12** or the number of CPUs **14** that are coupled to the bus, is not crucial to the operation of the present invention, and is not described in further detail below.

In accordance with this invention the system **10** further includes a Virtualized Application Program Subsystem (VAPS) **15**, shown for convenience as forming part of the CPU **14**. The VAPS **15** includes one or more interpreter programs capable of executing source programs, also referred to herein as macros, in the virtualized mode referred to above and described in further detail below. By example only, the source program may be a program that is suspected of containing a computer virus, or may be a "new" program that is to be tested to determine if it exhibits any viral characteristics or virus-type behavior. By defining a virtual environment within which the source program is to be interpreted, and then actually interpreting the source program in the virtual environment, the effect of the program's execution can be determined in a relatively "safe" mode, wherein any viral activity or characteristics can be readily ascertained. By example, if it is found that the source program unexpectedly attempts to access a virtualized mass storage medium, and/or to create one or more copies of itself in a region of virtualized memory, or in a virtual file, then the external program can be informed of the potentially viral nature of the source program.

It should be noted that the defined virtual environment need not correspond to the actual or real environment of the host computer system **10**. By example, the system **10** could be a multi-processor, high performance super-computer, while the defined virtual environment corresponds to a single processor, portable (laptop) personal computer. In a similar vein, the system **10** could be typical desktop single processor personal computer, while the simulated virtual environment may correspond to a multi-processor workstation.

The preferred software environment of the present invention (the VAPS **15** of FIG. **1**) is shown in FIG. **2**. The preferred software environment includes an application pro-

gram subsystem **100** and a virtual API **110**. At least one interpreter program **120** is provided to interface with a simulated or virtual environment **130**, a real environment **140**, a simulated or virtual application state **150**, and a real application state **160**. The real environment **140** will typically include a plurality of logical and physical devices such as files, a time-of-day (TOD) clock, an output device such as the display **20**, a manual input device such as a keyboard and/or a mouse, communication port(s), etc. The simulated environment **130** may include the same set of logical and physical devices as the real environment **140**, or a sub-set of these devices, or a different set of simulated devices altogether. As but one example, one of the simulated devices may be a special purpose interface to a machine that is controlled by the program being interpreted on a general purpose computer system **10** that does not include such an interface.

A program of interest, such as a macro written in a language that the interpreter is capable of interpreting, is passed through the virtual API **110** from an external program (i.e., a program that exists outside of the application program subsystem **100**.) The external program typically will also provide a specification of the simulated environment **130**, and may also provide a specification of the simulated application state **150**. Callback routines, notification criteria, and termination criteria, as described below, are also typically passed through the virtual API **110** from the external program. An output path **101** exists for providing indications of macro behavior from the application program subsystem **100** to the external program. The callback routines may use the output path **101**, which may form a part of the API **100**.

As but one example, a relatively simple data structure that may serve as a specification of a macro and its environment is as follows:

<a record containing a string with the name of the file to load the macro from; followed by a number indicating the maximum number of instructions the interpreter should execute before returning; followed by pointers to callback functions to be called when the macro attempts to read or write a file, or determine what files exist>.

The simulated application state can be specified in whatever way is deemed optimum for a given application. For example, callbacks would be passed as function pointers (or method references in an object-oriented system), and things such as virtual time and date are passed in a conventional format for representing the time and date, and certain parameters, such as those descriptive of how large the virtual hard disk would be, can be represented as integer numbers, and so on.

In an exemplary implementation of this invention, the application program subsystem **100** executes the steps as shown in FIG. **3**. In step **200**, a request for virtual execution is received by the application subsystem **100** via the API **110**. Next, in step **205**, the application program subsystem **100** locates and loads the macro to be simulated, and initializes the virtual or simulated environment **130**, all in accordance with information received with the API call of step **100**. The virtual environment **130** is a simulation of all the relevant parts of the real environment **140**, as well as any other required logical or physical devices. In this case variables and data areas in the virtual environment **130** represent resources such as the keyboard **22**, the display **20**, the time-of-day (TOD) clock, the file system, and any other system resource that a macro may read and/or attempt to alter. The virtual environment **130** may be initialized to a standard, default state, or it may be set according to parameters passed in from the external program via the virtual API **110**.

In step **210**, the application program subsystem **100** also initializes a simulated application state **150** which the macro will interact with, if one applies. The simulated application state **150** is a simulation of the real application state **160**. For example, and when simulating a word processing macro, a simulation of the applicable word processor application environment, having variables such as a current page, current line number, current font, etc., are initialized. After loading the macro to be simulated and initializing the virtual environment **130** and the virtual application state **150**, the interpreter **120** of the application program subsystem **100** begins in step **220** to run the macro by interpreting the current instruction (starting at the first instruction). At step **225** the interpreter **120** alters the simulated environment and the simulated application state as required by the execution of the instruction. For example, if the current program instruction reads a character from the simulated keyboard or the simulated I/O port, then a character is provided. If necessary, the simulated application state is changed as well. As but one example, if the newly inputted character would cause a simulated text buffer to exceed one full page of text, then the current page number of the simulated application state would be incremented by one, while the current line number would be reset to point to the first line of next page. In other words, whenever a macro instruction requires access to any resource that is being simulated, the subsystem **100** provides information to the macro from, or makes alterations to, the corresponding part of the virtual environment **130** instead, and if required the simulated application state is changed as well.

The application program subsystem **100** may also allow for callbacks to routines supplied via the API **110** from the external program. An external program might specify, for instance, that if the macro being simulated attempts to read data from a file (i.e. perform file I/O), a routine specified by the external program should be called (the callback routine) to determine what data the simulated read operation should return. In this case the external program that supplies the macro to be simulated, and the application program subsystem **100**, cooperate in simulating the virtual environment within which the macro is interpreted. In general, such conditions are checked in step **230**, wherein a determination is made if the execution of the current line of the macro being simulated has caused some external program notification criteria to be satisfied.

In general, an external program supplies a pointer to at least one callback routine through the API, and which may cause a termination of the virtual execution of the macro of interest by returning a particular result value.

In general, during each step of the simulation the application program subsystem **100** determines what interaction (s) the macro has had with the simulated environment **130**, and provides that information to the caller of the API **110**, either via callbacks as simulation proceeds (step **235**), or via a return code or other report when the simulation terminates (step **245**). The simulation terminates when a caller-specified condition occurs (e.g., some predetermined number of lines of code have been executed), when a callback function supplied by the caller indicates that simulation should terminate, or when the macro being simulated terminates. All of these conditions may be referred to as termination criteria, which are checked at step **240**. If the termination criteria are not satisfied, the control passes to step **250** to increment or otherwise change the macro instruction pointer, followed by a return to step **220** to interpret the next macro instruction that is now being pointed to.

As was made evident earlier, the macro being executed in the virtual environment and virtual application state may be suspected of harboring a macro virus, or may be a macro that is simply being exercised to verify that it does not cause any viral activity or manifestations. For example, if the execution of the macro results in a request to make a write access to disk or to main memory, then the external program can be made aware of the request through a callback, and can also be made aware of what data was actually written to the simulated disk or main memory. If the data that was written is discovered to be, for example, a copy of all or a portion of the macro, then this may be an indication of viral activity by the macro that is being interpreted in the virtual environment.

This invention thus provides a mechanism for an application program subsystem that includes a program interpreter, such as a macro interpreter, as well as an application program interface, to receive a macro to be interpreted through the API, as well as a specification of a virtual environment within which to execute the macro. The virtual environment specification could be provided as a data structure that specifies device names to be simulated, as well as characteristics of the simulated devices. The characteristics could be, by example, the size and address range of a memory buffer; the size, address range and interrupt numbers and levels associated with a disk; an I/O port type, address range, interrupt numbers and levels, and also protocol; the format of data output from a TOD clock device, as well as the address location of the device, etc. The virtual environment could as well be defaulted to be identical to the real environment of the computer system on which the application program subsystem and API reside. Through the use of callbacks the external program supplying the macro and virtual environment specification can be notified of every attempt by the macro to access a virtual resource, or the external program can be notified of a request to access only specific virtual resources or portions of a virtual resource (e.g., an attempt to write to or read from a specific range of main memory addresses or disk sectors). Any data written or read by the macro can also be captured and supplied to the external program for subsequent analysis.

In the simulation of resources, particularly those having real-time or time-critical characteristics, it is preferred that adjustments be made to accommodate for any changes in timing due to the simulation of the resource. For example, if the simulation of the TOD clock results in the clock actually running slower than the real TOD clock, then adjustments are made to bring the performance of the virtual TOD clock into agreement with the real TOD clock.

It should be realized that the APS **100**, as well as the interpreter **120**, the simulated environment **130**, the simulated application state, the virtual API **110**, as well as any other logical structures of interest, may all be provided as one or more computer programs that are embodied on a computer-readable medium, such as the hard disk **26**, the removable disk **30***a*, a volatile or a non-volatile memory, and/or any other suitable computer-readable medium.

While the invention has been particularly shown and described with respect to preferred embodiments of methods and apparatus thereof, it will be understood by those skilled in the art that changes in form and details may be made therein without departing from the scope and spirit of the invention.

What is claimed is:

1. A computer application program subsystem, comprising:

a program interpreter; and

an application program interface (API) through which an external program requests an execution of a program of interest in a specified simulated environment, the program of interest being written in a program language that the interpreter can interpret, said computer application program subsystem further comprising an output path for returning to the external program at least one indication of what action or actions the program of interest would have taken if the program of interest had been run in a real environment that corresponds to the specified simulated environment.

2. A computer application program subsystem as in claim **1**, wherein said output path is implemented using a callback function that is triggered upon the occurrence of an execution of an instruction of the program of interest satisfying at least one notification criterion.

3. A computer application program subsystem as in claim **1**, wherein the external program supplies a pointer to at least one callback routine through said API, and which can cause a termination of the virtual execution by returning a particular result value.

4. A computer application program subsystem as in claim **1**, wherein the external program that requests the execution of the program of interest further specifies a simulated application state.

5. A method for predicting the action of a program of interest within an application program subsystem having an interpreter, the method comprising the steps of:

receiving an application program interface (API) call for predicting the action of a program of interest;

initializing a virtual environment based on a real environment within which the program of interest is expected to operate;

initializing application state information;

interpreting in turn individual instructions of the program of interest; and

altering the application state information and the virtual environment in response to interpreted instructions.

6. A method as in claim **5**, wherein the step of receiving receives the API call from an external program, and further comprising a step of returning to the external program at least one indication of what action or actions the program of interest would have taken if the program of interest had been run in the real environment.

7. A method as in claim **5**, wherein the step of receiving receives the API call from an external program, and further comprising a step of executing a callback function to the external program for returning at least one indication of what action or actions the program of interest would have taken if the program of interest had been run in the real environment, the execution of the callback function being triggered upon the occurrence of an instruction of the program of interest satisfying at least one notification criterion.

8. A method as in claim **5**, wherein the step of receiving receives the API call from an external program, and further comprising a step of executing a callback function to the external program for returning at least one indication of what action or actions the program of interest would have taken if the program of interest had been run in the real environment, the execution of the callback function being triggered upon the occurrence of the program of interest satisfying a predetermined callback criterion.

9. A method for exercising a macro with an application program subsystem having a macro interpreter so as to detect a presence of potential viral activity, comprising steps of:

making an application program interface (API) call with a program, the API call identifying the macro and specifying an initial virtual environment within which the macro is to be interpreted;

interpreting in turn individual instructions of the macro;

altering the virtual environment in response to interpreted instructions; and

notifying the program upon the occurrence of an alteration to the virtual environment that triggers a predetermined notification criterion, wherein the alteration may be indicative of a presence of potential viral activity.

10. A method as in claim 9, and further comprising a step of notifying the program upon the occurrence of the interpretation of the macro triggering a predetermined termination criterion.

11. A method as in claim 9, wherein the API call further specifies an initial virtual application state within which the macro is to be interpreted.

12. A method as in claim 9, wherein the virtual environment is comprised of simulated logical and physical devices.

13. A method as in claim 9, wherein the method is executed on a host computer, and wherein the virtual environment is comprised of simulated physical and logical devices that are all present as real physical and logical devices in the host computer.

14. A method as in claim 9, wherein the method is executed on a host computer, and wherein the virtual environment is comprised of simulated physical and logical devices that are not all present as real physical and logical devices in the host computer.

15. A method for executing a macro with a data processing system, comprising the steps of:

generating an application program interface (API) call for interpreting a macro of interest in a virtual environment so as to execute all or a part of the macro of interest in the virtual environment;

the step of generating including a step of defining the virtual environment, wherein the virtual environment is defined at least in part through the use of at least one callback function that is invoked upon an occurrence of a virtual execution of the macro satisfying a predetermined callback function invocation criterion;

wherein the step of generating further includes a step of initializing state information;

interpreting in turn individual instructions of the macro; and

altering at least one of the state information and the virtual environment in response to interpreted instructions.

16. A method as in claim 15, wherein the predetermined callback function invocation criterion is comprised of an interpreted macro instruction attempting to perform file I/O.

17. A method as in claim 15, wherein the predetermined callback function invocation criterion is comprised of an interpreted macro instruction attempting to access memory.

18. A method as in claim 15, wherein the predetermined callback function invocation criterion is comprised of an interpreted macro instruction attempting to access a predetermined portion of a mass memory device.

19. A method as in claim 15, wherein the virtual environment is comprised of simulated physical and logical devices that are either all present as real physical and logical devices in the data processing system, or that are not all present as real physical and logical devices in the data processing system.

20. A computer program executable by a computer and embodied on a computer-readable medium for providing a computer application program subsystem, comprising:

a program interpreter code segment; and

an application program interface (API) code segment through which an external program requests an execution of a program of interest in a specified simulated environment, the program of interest being written in a program language that the interpreter code segment can interpret, said computer application program subsystem operating so as to return to the external program at least one indication of what action or actions the program of interest would have taken if the program of interest had been run in a real environment that corresponds to the specified simulated environment.

* * * * *

US006851057B1

(12) **United States Patent**
Nachenberg

(10) **Patent No.: US 6,851,057 B1**
(45) **Date of Patent: Feb. 1, 2005**

(54) **DATA DRIVEN DETECTION OF VIRUSES**

(75) Inventor: **Carey S. Nachenberg**, Northridge, CA (US)

(73) Assignee: **Symantec Corporation**, Cupertino, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/451,632**

(22) Filed: **Nov. 30, 1999**

(51) **Int. Cl.**[7] **G06F 11/30**; G06F 12/14; H04L 9/00; H04L 9/32

(52) **U.S. Cl.** **713/200**; 703/23; 703/26; 380/4; 380/25; 713/188; 713/201; 713/202; 914/38

(58) **Field of Search** 713/188, 200, 713/201; 914/38; 703/23, 26; 380/4, 25

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,386,523	A	* 1/1995	Crook et al.	711/220
5,696,822	A	* 12/1997	Nachenberg	713/200
5,796,989	A	8/1998	Morley et al.	
5,826,013	A	10/1998	Nachenberg	
5,854,916	A	12/1998	Nachenberg	
5,881,151	A	* 3/1999	Yamamoto	713/200
5,964,889	A	10/1999	Nachenberg	
5,999,723	A	12/1999	Nachenberg	
6,021,510	A	2/2000	Nachenberg	
6,067,410	A	5/2000	Nachenberg	
6,088,803	A	7/2000	Tso et al.	
6,094,731	A	7/2000	Waldin et al.	
6,311,277	B1	* 10/2001	Takaragi et al.	713/201
6,357,008	B1	3/2002	Nachenberg	

OTHER PUBLICATIONS

Symantec, Understanding Heuristics: Symantec's Bloodhound Technology, 1997, Symantec White Paper Series, vol. XXXIV.*

Trend Micro, Inc, Eliminating Viruses in the Lotus Notes Environment, 1999, Trend Micro Product Paper.*

Parkhouse, Jayne, "Pelican SafeTNet 2.0" [online], Jun. 2000, SC Magazine Product Review, [retrieved on Dec. 1, 2003]. Retrieved from the Internet: <URL: http://www.scmagazine.com/scmagazine/standalone/pelican/sc_pelican.html.

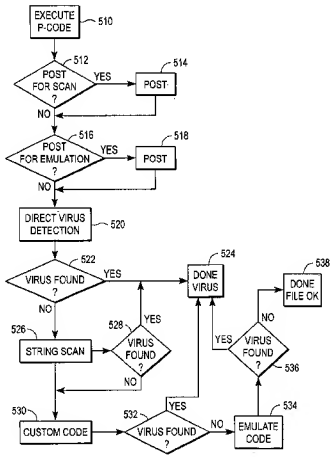
(List continued on next page.)

Primary Examiner—Ayaz Sheikh
Assistant Examiner—Shin-Hon Chen
(74) *Attorney, Agent, or Firm*—Fenwick & West LLP

(57) **ABSTRACT**

A virus detection system (VDS) **(400)** operates under the control of P-code to detect the presence of a virus in a file **(100)** having multiple entry points. P-code is an intermediate instruction format that uses primitives to perform certain functions related to the file **(100)**. The VDS **(400)** executes the P-code, which provides Turing-equivalent capability to the VDS. The VDS **(400)** has a P-code data file **(410)** for holding the P-code, a virus definition file (VDF) **(412)** for holding signatures of known viruses, and an engine **(414)** for controlling the VDS. The engine **(414)** contains a P-code interpreter **(418)** for interpreting the P-code, a scanning module **(424)** for scanning regions of the file **(100)** for the virus signatures in the VDF **(412)**, and an emulating module **(426)** for emulating entry points of the file. When executed, the P-code examines the file **(100)**, posts **(514)** regions that may be infected by a virus for scanning, and posts **(518)** entry points that may be infected by a virus for emulating. The P-code can also detect **(520)** certain viruses algorithmically. Then, the posted regions and entry points of the file **(100)** are scanned **(526)** and emulated **(534)** to determine if the file is infected with a virus. This technique allows the VDS **(400)** to perform sophisticated analysis of files having multiple entry points in a relatively brief amount of time. In addition, the functionality of the VDS **(400)** can be changed by changing the P-code, reducing the need for burdensome engine updates.

20 Claims, 5 Drawing Sheets



OTHER PUBLICATIONS

Padawer, "Microsoft P–Code Technology," [online]. Apr. 1992 [retrieved on Nov. 13, 2003]. Retrieved from the Internet: <: http://msdn.Microsoft.com/archive/en–us/dnarvc/html/msdn_c7pcode2.asp?frame=true.>, 6 pages.

"Frequently Asked Questions on Virus–L/comp.virus," [online]. Oct. 9, 1995 [retrieved on Nov. 25, 2003]. Retrieved from the Internet: <URL:http://www.claws–and–paws.com/virus/faqs/vlfaq200.shtml>, 53 pages.

LeCharlier et al., "Dynamic Detection and Classification of Computer Viruses Using General Behaviour Patterns," Proceedings of the Fifth International Virus Bulletin Conference, Boston, Mass., Sep. 20–22, 1995, 22 pages.

McCanne et al., "The BSD Packet Filter: A new Architecture for User–level Packet Capture," Preprint Dec. 19, 1992, 1993 Winter USENIX conference, San Diego, California, Jan. 25–29, 1993, 11 pages.

Leitold et al., "VIRus Searching and KILling Language," Proceedings of the Second International Virus Bulletin Conference, Sep. 1992, 15 pages.

Taubes, "An Immune System for Cyberspace," Think Research [online], vol. 34, No. 4, 1996 [retrieved on Dec. 15, 2003]. Retrieved from the Internet: <URL: http://domino.research.ibm.com/comm./wwwr_thinkresearch.nsf/pages/antivirus496.html>, 9 pages.

Ször, "Memory Scanning Under Windows NT," Virus Bulletin Conference, Sep. 1999, 22 pages.

Ször, "Attacks on Win32," Virus Bulletin Conference, Oct. 1998, 84 pages.

* cited by examiner

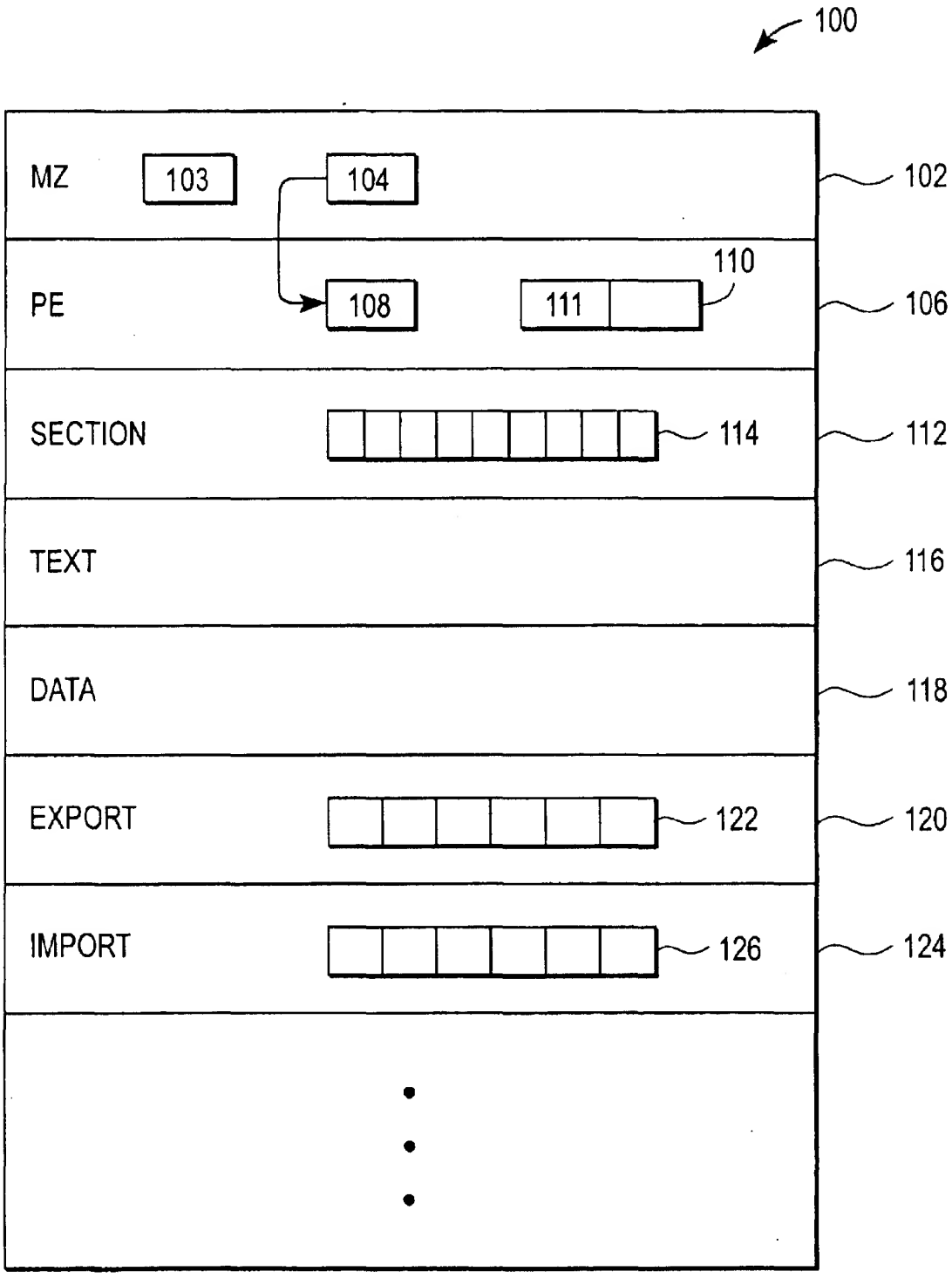


FIG. 1 (PRIOR ART)

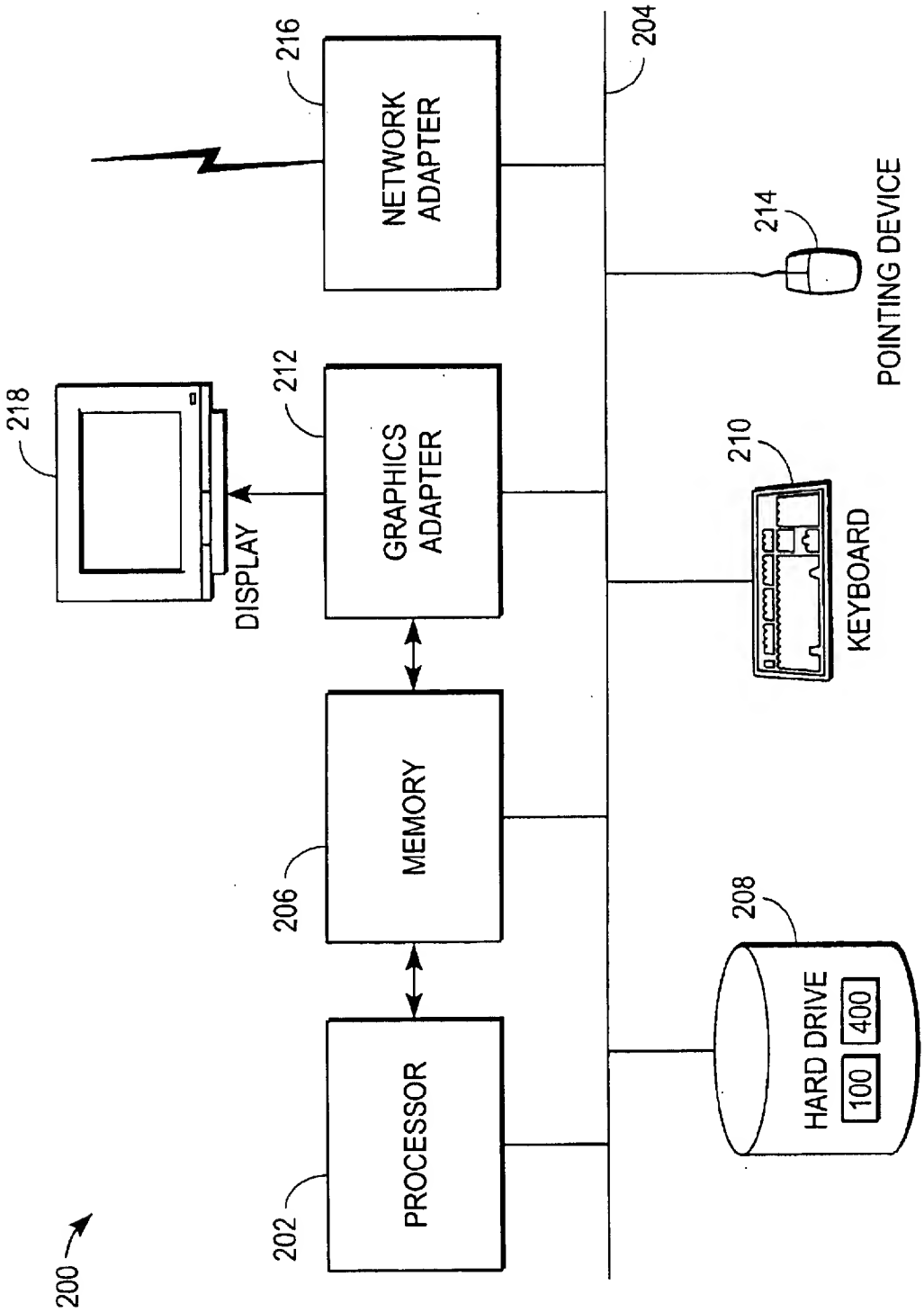

200
PROCESSOR
202
MEMORY
206
GRAPHICS ADAPTER
212
NETWORK ADAPTER
216
204
HARD DRIVE
100
400
208
KEYBOARD
210
POINTING DEVICE
214
DISPLAY
218

FIG. 2

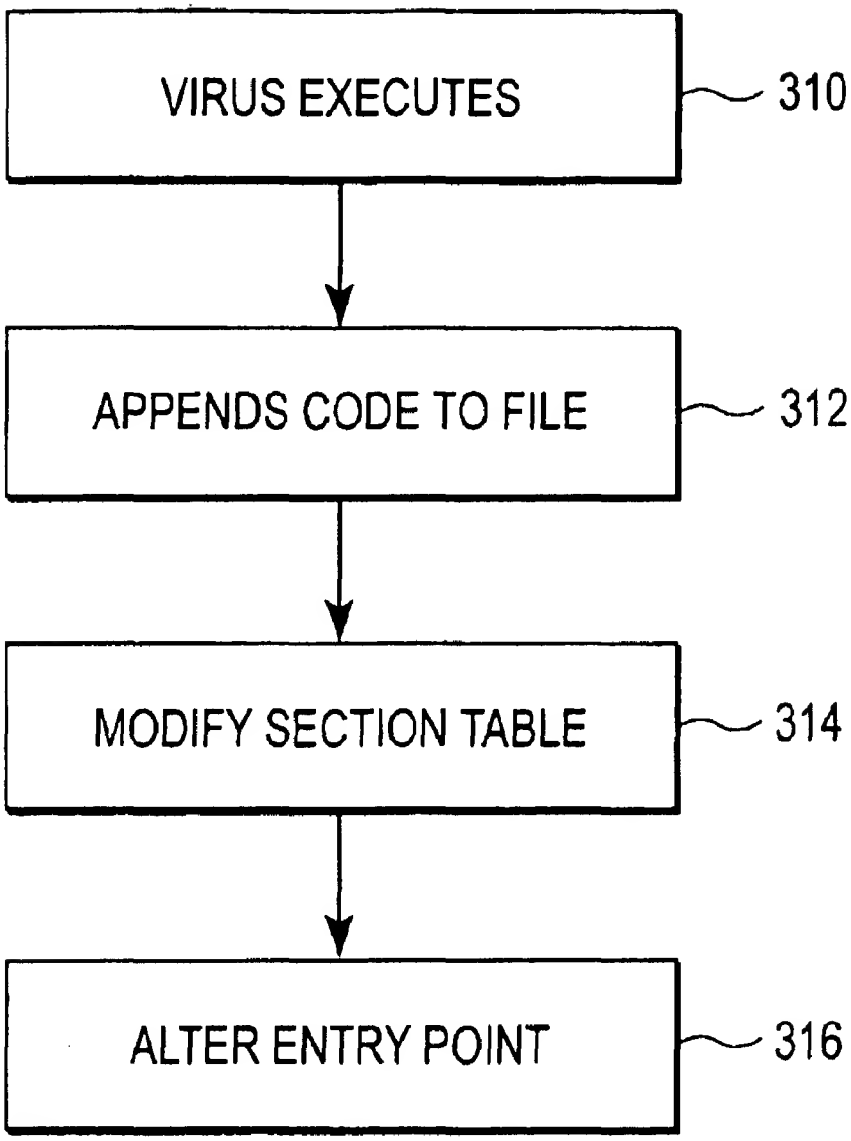

VIRUS EXECUTES
310
APPENDS CODE TO FILE
312
MODIFY SECTION TABLE
314
ALTER ENTRY POINT
316

FIG. 3

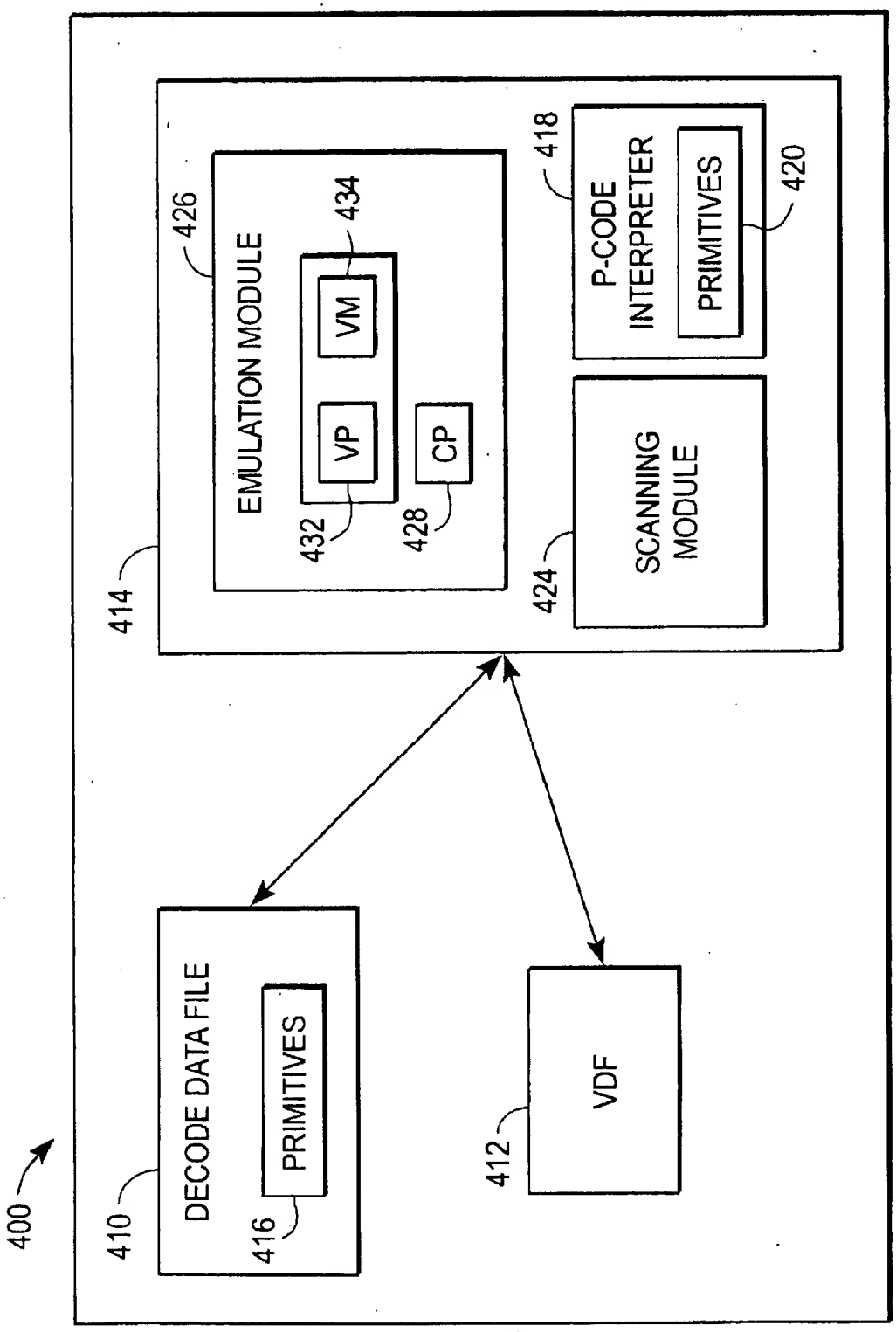

400
410
DECODE DATA FILE
416
PRIMITIVES
412
VDF
414
426
EMULATION MODULE
434
VM
VP
CP
432
428
418
P-CODE INTERPRETER
PRIMITIVES
420
424
SCANNING MODULE

FIG. 4

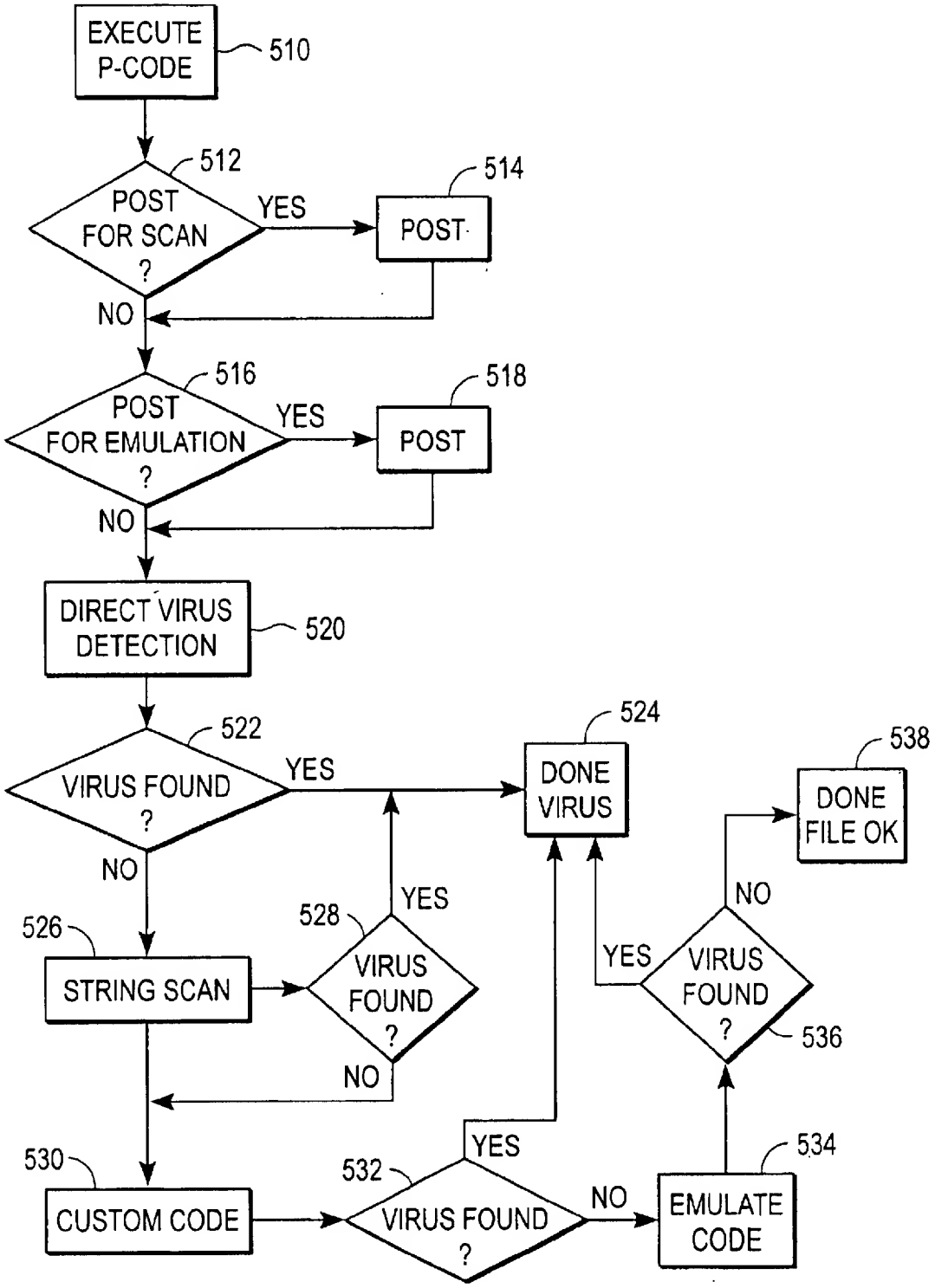

EXECUTE P-CODE
510
512
POST FOR SCAN ?
YES
514
POST
NO
516
POST FOR EMULATION ?
YES
518
POST
NO
DIRECT VIRUS DETECTION
520
522
VIRUS FOUND ?
YES
524
DONE VIRUS
538
DONE FILE OK
NO
526
STRING SCAN
528
VIRUS FOUND ?
YES
NO
VIRUS FOUND ?
YES
NO
536
530
CUSTOM CODE
532
VIRUS FOUND ?
YES
NO
534
EMULATE CODE

FIG. 5

DATA DRIVEN DETECTION OF VIRUSES

BACKGROUND

FIELD OF THE INVENTION

This invention pertains in general to detecting viruses within files in digital computers and more particularly to detecting the presence of a virus in a file having multiple entry points.

BACKGROUND OF THE INVENTION

Simple computer viruses work by copying exact duplicates of themselves to each executable program file they infect. When an infected program executes, the simple virus gains control of the computer and attempts to infect other files. If the virus locates a target executable file for infection, it copies itself byte-for-byte to the target executable file. Because this type of virus replicates an identical copy of itself each time it infects a new file, the simple virus can be easily detected by searching in files for a specific string of bytes (i.e. a "signature") that has been extracted from the virus.

Encrypted viruses comprise a decryption routine (also known as a decryption loop) and an encrypted viral body. When a program file infected with an encrypted virus executes, the decryption routine gains control of the computer and decrypts the encrypted viral body. The decryption routine then transfers control to the decrypted viral body, which is capable of spreading the virus. The virus is spread by copying the identical decryption routine and the encrypted viral body to the target executable file. Although the viral body is encrypted and thus hidden from view, these viruses can be detected by searching for a signature from the unchanging decryption routine.

Polymorphic encrypted viruses ("polymorphic viruses") comprise a decryption routine and an encrypted viral body which includes a static viral body and a machine-code generator often referred to as a "mutation engine." The operation of a polymorphic virus is similar to the operation of an encrypted virus, except that the polymorphic virus generates a new decryption routine each time it infects a file. Many polymorphic viruses use decryption routines that are functionally the same for all infected files, but have different sequences of instructions.

These multifarious mutations allow each decryption routine to have a different signature. Therefore, polymorphic viruses cannot be detected by simply searching for a signature from a decryption routine. Instead, antivirus software uses emulator-based antivirus technology, also known as Generic Decryption (GD) technology, to detect the virus. The GD scanner works by loading the program into a software-based CPU emulator which acts as a simulated virtual computer. The program is allowed to execute freely within this virtual computer. If the program does in fact contain a polymorphic virus, the decryption routine is allowed to decrypt the viral body. The GD scanner can then detect the virus by searching through the virtual memory of the virtual computer for a signature from the decrypted viral body.

Metamorphic viruses are not encrypted but vary the instructions in the viral body with each infection of a host file. Accordingly, metamorphic viruses often cannot be detected with a string search because they do not have static strings.

Regardless of whether the virus is simple, encrypted, polymorphic, or metamorphic, the virus typically infects an executable file by attaching or altering code at or near an "entry point" of the file. An "entry point" is an instruction or instructions in the file that a virus can modify to gain control of the computer system on which the file is being executed. Many executable files have a "main entry point" containing instructions that are always executed when the program is invoked. Accordingly, a virus seizes control of the program by manipulating program instructions at the main entry point to call the virus instead of the program. The virus then infects other files on the computer system.

When infecting a file, the virus typically stores the viral body at the main entry point, at the end of the program file, or at some other convenient location in the file. When the virus completes execution, it calls the original program instructions that were altered by the virus.

In order to detect the presence of a virus, antivirus software typically scans the code near the main entry point, and other places where the viral body is likely to reside, for strings matching signatures held in a viral signature database. In addition, the antivirus software emulates the code near the main entry point in an effort to decrypt any encrypted viral bodies. Since viruses usually infect only the main entry point, the antivirus software can scan and emulate a file relatively quickly. When new viruses are detected, the antivirus software can be updated by adding the new viral signatures to the viral signature database.

More recently, however, viruses have been introduced that infect entry points other than the main entry point. As a result, the number of potential entry points for a viral infection in a typical search space, such as a MICROSOFT WINDOWS portable executable (PE) file, is very large. Prior art antivirus software would require an extremely long processing time to scan and/or emulate the code surrounding all of the entry points in the file that might be infected by a virus.

Moreover, the multiple entry points provide opportunities for viruses to use previously unknown methods to infect a file. As a result, it may not be possible to detect the virus merely by adding a new signature to the viral signature database. In many cases, the virus detection system itself must be updated with hand-coded virus detection routines in order to detect the new viruses. Writing custom detection routines and updating the antivirus software requires a considerable amount of work, especially when the antivirus software is distributed to a mass market.

Therefore, there is a need in the art for antivirus software that can detect viruses in PE and other files having multiple entry points without requiring a prohibitively large amount of processing time. There is also a need that the antivirus software be easily upgradeable, so that new virus detection capabilities can be added without requiring hand-coded virus detection logic or needing to distribute a new virus detection engine.

SUMMARY OF THE INVENTION

The above needs are met by a virus detection system (VDS) **(400)** for detecting the presence of a virus in a file **(100)** having multiple entry points. The VDS **(400)** preferably includes a data file **(410)** holding P-code instructions. P-code is an interpreted language that provides the VDS **(400)** with Turing machine-equivalent behavior, and allows the VDS to be updated by merely updating the P-code. The VDS **(400)** also includes a virus definition file (VDF) **(412)** containing virus signatures for known viruses. Each virus signature is a string of bytes characteristic of the static viral body of the given virus.

The VDS (**400**) is controlled by an engine (**414**) having a P-code interpreter (**418**) for interpreting the P-code in the data file (**410**). The P-code interpreter (**418**) may also contain primitives (**420**) that can be invoked by the P-code. Primitives are functions that can be called by the P-code. The primitives (**420**) preferably perform file and memory manipulations, and can also perform other useful tasks. In addition, the engine (**414**) has a scanning module (**424**) for scanning a file or range of memory for virus signatures in the VDF (**412**) and an emulating module (**426**) for emulating code in the file (**100**) in order to decrypt polymorphic viruses and detect the presence of metamorphic viruses.

The engine (**414**) interprets the P-code in the P-code data file (**410**) and responds accordingly. In one embodiment, the P-code examines the entry points in the file (**100**) to determine whether the entry points might be infected with a virus. Those entry points and other regions of the file (**100**) commonly infected by viruses or identified by suspicious characteristics in the file, such as markers left by certain viruses, are posted (**514**) for scanning. Likewise, the P-code posts (**518**) entry points and starting contexts for regions of the file (**100**) that are commonly infected by viruses or bear suspicious characteristics for emulating. Using the P-code to preprocess regions of the file (**100**) and select only those regions or entry points that are likely to contain a virus for subsequent scanning and/or emulating allows the VDS (**400**) to examine files for viruses that infect places other than the main entry point in a reasonable amount of time. The P-code can also determine whether the file (**100**) is infected with a virus by using virus detection routines written directly into the P-code, thereby eliminating the need to scan for strings or emulate the file (**100**).

A region posted for string scanning is identified by a range of memory addresses. Preferably, the P-code merges postings having overlapping ranges so that a single posting specifies the entire region to be scanned. When an entry point is posted for emulating, the P-code specifies the emulation context, or starting state to be used for the emulation. An entry point can be posted multiple times with different contexts for each emulation.

The engine (**414**) uses the scanning module (**424**) to scan the regions of the file (**100**) that are posted for scanning by the P-code for the virus signatures in the VDF (**412**). If the scanning module (**424**) detects a virus, the VDS (**400**) preferably reports that the file (**100**) is infected and stops operation.

If the scanning module (**424**) does not find a virus in the posted regions, a preferred embodiment of the present invention optionally utilizes a hook to call (**530**) custom virus detection code. The hook allows virus detection engineers to insert a custom program into the VDS (**400**) and detect viruses that, for reasons of speed and efficiency, are better detected by custom code.

Then, the VDS (**400**) preferably uses the emulating module (**426**) to emulate the posted entry points. Preferably, each posted entry point is emulated for enough instructions to allow polymorphic and metamorphic viruses to decrypt or otherwise become apparent. Once emulation is complete, the VDS (**400**) uses the scanning module (**424**) to scan pages of the virtual memory (**434**) that were either modified or emulated through for signatures of polymorphic viruses and uses stochastic information obtained during the emulation, such as instruction usage profiles, to detect metamorphic viruses. If the scanning module (**424**) or VDS (**400**) detects a virus, the VDS reports that the file (**100**) is infected. Otherwise, the VDS (**400**) reports that it did not detect a virus in the file (**100**).

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a high-level block diagram of a conventional executable file **100** having multiple entry points that can be infected by a virus;

FIG. **2** is a high-level block diagram of a computer system **200** for storing and executing the file **100** and a virus detection system (VDS) **400**;

FIG. **3** is a flow chart illustrating steps performed by a typical virus when infecting the file **100**;

FIG. **4** is a high-level block diagram of the VDS **400** according to a preferred embodiment of the present invention; and

FIG. **5** is a flow chart illustrating steps performed by the VDS **400** according to a preferred embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

In order to accomplish the mischief for which they are designed, software viruses must gain control of a computer's central processing unit (CPU). Viruses typically gain this control by attaching themselves to an executable file (the "host file") and modifying the executable image of the host file at an entry point to pass control of the CPU to the viral code. The virus conceals its presence by passing control back to the host file after it has run by calling the original instructions at the modified entry point.

Viruses use different techniques to infect the host file. For example, a simple virus always inserts the same viral body into the target file. An encrypted virus infects a file by inserting an unchanging decryption routine and an encrypted viral body into the target file. A polymorphic encrypted virus (a "polymorphic virus") is similar to an encrypted virus, except that a polymorphic virus generates a new decryption routine each time it infects a file. A metamorphic virus is not encrypted, but it reorders the instructions in the viral body into a functionally equivalent, but different, virus each time it infects a file. Simple and encrypted viruses can typically be detected by scanning for strings in the viral body or encryption engine, respectively. Since polymorphic and metamorphic viruses usually do not have static signature strings, polymorphic and metamorphic viruses can typically be detected by emulating the virus until either the static viral body is decrypted or the virus otherwise becomes apparent. While this description refers to simple, encrypted, polymorphic, and metamorphic viruses, it should be understood that the present invention can be used to detect any type of virus, regardless of whether the virus fits into one of the categories described above.

A virus typically infects an executable file by attaching or altering code at or near an entry point of the file. An "entry point" is any instruction or instructions in the file that a virus can modify to gain control of the computer system on which the file is being executed. An entry point is typically identified by an offset from some arbitrary point in the file. Certain entry points are located at the beginning of a file or region and, thus, are always invoked when the file or region is executed. For example, an entry point can be the first instruction executed when a file is executed or a function within the file is called. Other entry points may consist of single instructions deep within a file that can be modified by a virus. For example, the entry point can be a CALL or JMP instruction that is modified to invoke viral code. Once a virus seizes control of the computer system through the entry point, the virus typically infects other files on the system.

FIG. **1** is a high-level block diagram of an executable file **100** having multiple entry points that can be infected by a virus as described above. In the example illustrated by FIG. **1**, the executable file is a Win32 portable executable (PE) file intended for use with a MICROSOFT WINDOWS-based operating system (OS), such as WINDOWS 98, WINDOWS NT, and WINDOWS 2000. Typically, the illustrated file **100** is of the type .EXE, indicating that the file is an executable file, or .DLL, indicating that the file is a dynamic link library (DLL). However, the present invention can be used with any file, and is not limited to only the type of file illustrated in FIG. **1**. APPLE MACINTOSH files, for example, share many similarities with Win32 files, and the present invention is equally applicable to such files.

The file **100** is divided into sections containing either code or data and aligned along four kilobyte (KB) boundaries. The MS-DOS section **102** contains the MS-DOS header **102** and is marked by the characters "MZ". This section **102** contains a small executable program **103** designed to display an error message if the executable file is run in an unsupported OS (e.g., MS-DOS). This program **103** is an entry point for the file **100**. The MS-DOS section **102** also contains a field **104** holding the relative offset to the start **108** of the PE section **106**. This field **104** is another entry point for the file **100**.

The PE section **106** is marked by the characters "PE" and holds a data structure **110** containing basic information about the file **100**. The data structure **110** holds many data fields describing various aspects of the file **100**. One such field is the "checksum" field **111**, which is rarely used by the OS.

The next section **112** holds the section table **114**. The section table **114** contains information about each section in the file **100**, including the section's type, size, and location in the file **100**. For example, entries in the section table **114** indicate whether a section holds code or data, and whether the section is readable, writeable, and/or executable. Each entry in the section table **114** describes a section that may have multiple, one, or no entry points.

The text section **116** holds general-purpose code produced by the compiler or assembler. The data section **118** holds global and static variables that are initialized at compile time.

The export section **120** contains an export table **122** that identifies functions exported by the file **100** for use by other programs. An EXE file might not export any functions but DLL files typically export some functions. The export table **122** holds the function names, entry point addresses, and export ordinal values for the exported functions. The entry point addresses typically point to other sections in the file **100**. Each exported function listed in the export table **122** is an entry point into the file **100**.

The import section **124** has an import table **126** that identifies functions that are imported by the file **100**. Each entry in the import table **126** identifies the external DLL and the imported function by name. When code in the text section **116** calls a function in another module, such as an external DLL file, the call instruction transfers control to a JMP instruction also in the text section **116**. The JMP instruction, in turn, directs the call to a location within the import table **126**. Both the JMP instruction and the entries in the import table **126** represent entry points into the file **100**. Additional information about the Win32 file format is found in M. Pietrek, "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format," Microsoft Systems Journal, March 1994, which is hereby incorporated by reference.

FIG. **2** is a high-level block diagram of a computer system **200** for storing and executing the host file **100** and a virus detection system (VDS) **400**. Illustrated are at least one processor **202** coupled to a bus **204**. Also coupled to the bus **204** are a memory **206**, a storage device **208**, a keyboard **210**, a graphics adapter **212**, a pointing device **214**, and a network adapter **216**. A display **218** is coupled to the graphics adapter **212**.

The at least one processor **202** may be any general-purpose processor such as an INTEL x86, SUN MICROSYSTEMS SPARC, or POWERPC compatible-CPU. The storage device **208** may be any device capable of holding data, like a hard drive, compact disk read-only memory (CD-ROM), DVD, or a solid-state memory device. The memory **206** holds instructions and data used by the processor **202**. The pointing device **214** may be a mouse, track ball, light pen, touch-sensitive display, or other type of pointing device, and is used in combination with the keyboard **210** to input data into the computer system **200**. The graphics adapter **212** displays images and other information on the display **218**. The network adapter **216** couples the computer system **200** to a local or wide area network.

Preferably, the host file **100** and program modules providing the functionality of the VDS **400** are stored on the storage device **208**. The program modules, according to one embodiment, are loaded into the memory **206** and executed by the processor **202**. Alternatively, hardware or software modules for providing the functionality of the VDS **400** may be stored elsewhere within the computer system **200**.

FIG. **3** is a flow chart illustrating steps performed by a typical virus when infecting the host file **100**. The illustrated steps are merely an example of a viral infection and are not representative of any particular virus. Initially, the virus executes **310** on the computer system **200**. The virus may execute, for example, when the computer system **200** executes or calls a function in a previously-infected file.

When the host file **100** is opened, the virus appends **312** the viral code to a location within the file. For example, the virus can append the viral body to the slack space at the end of a section or put the viral body within an entirely new section. The virus can be, for example, simple, encrypted, polymorphic, or metamorphic.

The virus also modifies **314** the section table **114** to account for the added viral code. For example, the virus may change the size entry in the section table **114** to account for the added viral code. Likewise, the virus may add entries for new sections added by the virus. If necessary, the virus may mark an infected section as executable and/or place a value in a little used field, such as the checksum field **111**, to discreetly mark the file as infected and prevent the virus from reinfecting the file **100**.

In addition, the virus alters **316** an entry point of the file **100** to call the viral code. The virus may accomplish this step by, for example, overwriting the value in the field **104** holding the relative offset to the start **108** of the PE section **106** with the relative offset to virus code stored elsewhere in the file. Alternatively, the virus can modify entries in the export table **122** to point to sections of virus code instead of the exported functions. A virus can also modify the destination of an existing JMP or CALL instruction anywhere in the file **100** to point to the location of viral code elsewhere in the file, effectively turning the modified instruction into a new entry point for the virus.

FIG. **4** is a high-level block diagram of the VDS **400** according to a preferred embodiment of the present invention. The VDS **400** includes a P-code data file **410**, a virus

definition file (VDF) **412**, and an engine **414**. The P-code data file **410** holds P-code instructions for examining the host file **100**. As used herein, "P-code" refers to program code instructions in an interpreted computer language. The P-code provides a Turing-equivalent programmable system which has all of the power of a program written in a more familiar language, such as C. Preferably, the P-code instructions in the data file **410** are created by writing instructions in any computer language and then compiling the instructions into P-code. Other portable, i.e., cross-platform, languages or instruction representations, such as JAVA, may be used as well.

The VDF **412** preferably holds an entry or virus definition for each known virus. Each virus definition contains information specific to a virus or strain of viruses, including a signature for identifying the virus or strain. An entry in the VDF **412**, according to an embodiment of the present invention, is organized as follows:

[VirusID]
0x2f41
[SigStart]
0x89, 0xb4, 0xb8, 0x02, 0x096, 0x56, DONE
[SigEnd]

Here, [VirusID] is a data field for a number that identifies the specific virus or virus strain. [SigStart] and [SigEnd] bracket a virus signature, which is a string of bytes characteristic of the virus or strain having Virus ID 0x2f41. The signature, for example, may identify the static encryption engine of an encrypted virus or the static viral body of a polymorphic virus. The virus signatures are used to detect the presence of a virus in a file (or in the virtual memory **434** after emulating), typically by performing a string scan for the bytes in the signature. In one embodiment of the present invention, the VDF **412** holds virus definitions for thousands of viruses.

The engine **414** controls the operation of the VDS **400**. The engine **414** preferably contains a P-code interpreter **418** for interpreting the P-code in the P-code data file **410**. The interpreted P-code controls the operation of the engine **414**. In alternative embodiments where the data file **410** holds instructions in a format other than P-code, the engine **414** is equipped with a module for interpreting or compiling the instructions in the relevant format. For example, if the data file **410** holds JAVA instructions, the engine **414** preferably includes a JAVA Just-in-Time compiler.

The P-code interpreter **418** preferably includes special P-code function calls called "primitives" **420**. The primitives **420** can be, for example, written in P-code or a native language, and/or integrated into the interpreter itself. Primitives **420** are essentially functions useful for examining the host file **100** and the virtual memory **434** that can be called by other P-code. For example, the primitives **420** perform functions such as opening files for reading, closing files, zeroing out memory locations, truncating memory locations, locating exports in the file, determining the type of the file, and finding the offset of the start of a function. The functions performed by the primitives **420** can vary depending upon the computer or operating system in which the VDS **400** is being used. For example, different primitives may be utilized in a computer system running the MACINTOSH operating system than in a computer system running a version of the WINDOWS operating system. In an alternative embodiment, some or all of the primitives **416** can be stored in the P-code data file **410** instead of the interpreter **418**.

The engine **414** also contains a scanning module **424** for scanning pages of the virtual memory **434** or regions of a file **100** for virus signatures held in the VDF **412**. In one embodiment, the scanning module **424** receives a range of memory addresses as parameters. The scanning module scans the memory addresses within the supplied range for signatures held in the VDF **412**.

The engine **414** also contains an emulating module **426** for emulating code in the file **100** starting at an entry point. The emulating module includes a control program **428** for setting up a virtual machine **430** having a virtual processor **432** and an associated virtual memory **434**. The virtual machine can emulate a 32-bit MICROSOFT WINDOWS environment, an APPLE MACINTOSH environment, or any other environment for which emulation is desired. The virtual machine **430** uses the virtual processor **432** to execute code in the virtual memory **434** in isolation from the remainder of the computer system **200**. Emulation starts with a given context, which specifies the contents of the registers, stacks, etc. in the virtual processor **432**. During emulation, every page of virtual memory **434** that is read from, written to, or emulated through is marked. The number of instructions that the virtual machine **430** emulates can be fixed at the beginning of emulation or can be determined adaptively while the emulation occurs.

FIG. **5** is a flow chart illustrating steps performed by the VDS **400** according to a preferred embodiment of the present invention. The behavior of the VDS **400** is controlled by the P-code. Since the P-code provides Turing machine-like functionality to the VDS **400**, the VDS **400** has an infinite set of possible behaviors. Accordingly, it should be understood that the steps illustrated in FIG. **5** represent only one possible set of VDS **400** behaviors.

Initially, the engine **414** executes **510** the P-code in the P-code data file **410**. Next, the P-code determines **512** which areas of the file **100** should be scanned for virus strings because the areas are likely to contain a simple or encrypted virus. Areas of the file **100** that should be scanned are posted **514** for later scanning. Typically, the main entry point of the PE header and the last section of the file **100** are always posted **514** for string scanning because these are the areas most likely to be infected by a virus. Any other region of the file can be posted **514** for scanning if the regions seem suspicious. For example, if the destination of a JMP or CALL instruction points to a suspicious location in the file **100**, it may be desirable to post the areas of the file surrounding both the instruction and the destination.

For other regions of the file **100**, the determination of whether to scan is made based on tell-tale markers set by the viruses, such as unusual locations and lengths of sections, or unusual attribute settings of fields within the sections. For example, if the value of an unused field, such as the checksum field **111**, is set or the length of a section is suspiciously long, then the P-code posts **514** a region of the section for scanning. Likewise, if a section that is normally not executable is marked as executable, then the P-code preferably posts **514** a region of the section for scanning.

Next, the P-code determines **516** which entry points should be posted **518** for emulating because the entry points are likely to execute polymorphic or metamorphic viruses. The P-code checks the main entry point **103** for known non-viral code. If such code is not found, then the P-code posts the main entry point **103** for emulating. Entry points in other regions of the file **100** are posted **518** for emulating if the code exhibits evidence of viral infection. For example, an entry point in a region of the file **100** is preferably posted for emulating if the checksum field **111** in the header contains a suspicious value. When an entry point is posted for emulating, an emulation context, or starting state of the computer system **200**, is also specified.

The P-code can also identify **520** viruses in the file **100** without emulating or string searching. This identification is performed algorithmically or stochastically using virus definitions written into the P-code. The virus definitions preferably use the primitives **420** in the interpreter **418** to directly test the file **100** for characteristics of known viruses. For example, if the last five bytes of a file or section have a certain signature found in only one virus, or the file size is evenly divisible by **10**, characteristics likely to occur only if the file is infected by certain viruses, then the P-code can directly detect the presence of the virus. In addition, the P-code can be enhanced with algorithms and heuristics to detect the behavior of unknown viruses. If a virus is found **522** by the P-code, the VDS **400** can stop **524** searching and report that the file **100** is infected with a virus.

Scan requests posted by the P-code are preferably merged and minimized to reduce redundant scanning. For instance, a posted request to scan bytes **1000** to **1500**, and another posted request to scan bytes **1200** to **3000**, are preferably merged into a single request to scan bytes **1000** to **3000**. Any merging algorithm known to those skilled in the art can be used to merge the scan requests. Posted emulating requests having identical contexts can also be merged, although such posts occur less frequently than do overlapping scan requests.

If the P-code does not directly detect **522** a virus, the VDS **400** next preferably performs scans on the posted regions of the file **100**. The VDS **400** executes **526** the scanning module **424** to scan the posted regions for the virus signatures of simple and encrypted viruses found in the VDF **412**. If a virus is found **528** by the scanning module **424**, the VDS **400** stops scanning **524** and reports that the file **100** is infected with a virus.

If neither the P-code nor the scanning module **424** detects the presence of a virus, the VDS **400** preferably utilizes a hook to execute **530** custom virus-detection code. The hook allows virus detection engineers to insert custom virus detection routines written in C, C++, or any other language into the VDS **400**. The custom detection routines may be useful to detect unique viruses that are not practical to detect via the P-code and string scanning. For example, it may be desired to use faster native code to detect a certain virus rather than the slower P-code. Alternate embodiments of the present invention may provide hooks to custom code at other locations in the program flow. If a virus is found **532** by the custom code, the VDS **400** can stop searching **524** for a virus and report that the file **100** is infected.

If the P-code, scanning module **424**, and custom code fail to detect a virus, the VDS **400** preferably executes the emulating module **426**. The emulating module **426** emulates **534** the code at the entry point posted by the P-code in order to decrypt polymorphic viruses and trace through code to locate metamorphic viruses. Once enough instructions have been emulated that any virus should become apparent (i.e., a polymorphic virus has decrypted the static viral body or the code of a metamorphic virus is recognized), the emulating module **426** preferably detects a polymorphic virus by using the scanning module **424** to scan pages of virtual memory **434** that were marked as modified or executed through for any virus signatures. The emulation module **426** preferably detects a metamorphic virus via stochastic information obtained during emulation, such as instruction usage profiles. If **536** a virus is found **534** by the emulating module **426**, the VDS **400** reports that the file **100** is infected. Otherwise, the VDS **400** reports **538** that it did not detect a virus in the file **100**.

In sum, the VDS **400** according to the present invention uses P-code and primitives **420** to extend the possible behaviors of the VDS. The P-code also allows the VDS **400** to be updated to detect new viruses without costly engine upgrades. In addition, the behavior of the VDS **400** is adapted to examine files having multiple entry points in a reasonable amount of time.

The above description is included to illustrate the operation of the preferred embodiments and is not meant to limit the scope of the invention. The scope of the invention is to be limited only by the following claims. From the above discussion, many variations will be apparent to one skilled in the relevant art that would yet be encompassed by the spirit and scope of the invention.

I claim:

1. A virus detection system for detecting if a computer file is infected by a virus, the file having a plurality of potential virus entry points, the system comprising:

an engine for controlling operation of the virus detection system responsive to instructions stored in an intermediate language, the instructions adapted to examine the plurality of potential virus entry points and post for emulating ones of the plurality of potential virus entry points exhibiting characteristics indicating a possible virus;

an emulating module coupled to the engine for emulating the posted entry points of the file in a virtual memory responsive to the engine, wherein the virus may become apparent during the emulation of an entry points of the file infected by the virus; and

a scanning module coupled to the engine for scanning regions of the virtual memory for a signature of the virus responsive to the engine and the emulating module, wherein presence of the virus signature in a scanned region indicates that the file is infected by the virus.

2. The virus detection system of claim **1**, further comprising:

a custom module coupled to the engine for executing custom virus-detection code responsive to invocation by the engine.

3. The virus detection system of claim **1**, wherein the intermediate language is P-code and the engine comprises:

a P-code interpreter for interpreting the P-code and controlling the operation of the virus detection system responsive thereto.

4. The virus detection system of claim **3**, wherein the engine further comprises:

primitives for performing operations with respect to the file and the virtual memory responsive to invocations of the primitives by the P-code.

5. The virus detection system of claim **1**, further comprising:

a virus definition file coupled to the scanning module for holding virus signatures for use by the scanning module.

6. The virus detection system of claim **1**, wherein the instructions stored in the intermediate language post regions of the file for scanning by the scanning module.

7. The virus detection system of claim **6**, wherein postings identifying overlapping regions are merged into a single posting identifying the regions of the merged postings.

8. A method for detecting a virus in a computer file, the file having a plurality of potential virus entry points, the method comprising the steps of:

executing instructions stored in an intermediate language representation, the instructions performing the steps of:

examining regions of the file for possible infection by viruses and posting for scanning any regions exhibiting characteristics indicating a possible virus infection;

examining the plurality of potential virus entry points of the file for possible infections by viruses and posting for emulating ones of the plurality of potential virus entry points exhibiting characteristics indicating a possible virus infection; and

examining the posted regions of the file to algorithmically determine whether the file is infected with a virus.

9. The method of claim **8**, wherein the instructions further perform the steps of:

merging overlapping regions posted for scanning.

10. The method of claim **8**, wherein the instructions further perform the step of:

calling a custom executable program to determine when the file is infected with a virus.

11. The method of claim **8**, further comprising the step of:

scanning the regions of the file posted for scanning for signatures of known viruses.

12. The method of claim **8**, further comprising the steps of:

emulating the posted entry points in a virtual memory to allow the viruses to become apparent;

scanning the virtual memory for signatures of the viruses; and

examining stochastic information obtained during emulation to detect the presence of the known viruses.

13. The method of claim **8**, wherein the step of examining the plurality of potential virus entry points of the file for possible infections by viruses and posting for emulating ones of the plurality of potential virus entry points exhibiting characteristics indicating a possible virus infection comprises the step of:

determining if a main entry point of the file has known non-viral code;

wherein the main entry point is posted for emulating responsive to a determination that the main entry point does not have known non-viral code.

14. A computer program product comprising:

a computer usable medium having computer readable code embodied therein for determining if a computer file is infected by a virus, the file having a plurality of potential virus entry points, the computer readable code comprising:

an engine for controlling the operation of the computer program product responsive to instructions stored in an intermediate language, the instructions adapted to examine the plurality of potential virus entry points and post for emulating ones of the plurality of potential virus entry points exhibiting characteristics indicating a possible virus infection;

an emulating module for emulating the posted entry points of the file in a virtual memory responsive to the engine, wherein the virus may become apparent during emulation of an entry points of the file infected by the virus; and

a scanning module for scanning regions of the virtual memory for a signature of the virus responsive to the engine and the emulating module, wherein presence of the virus signature indicates that the file is infected by the virus.

15. The computer program product of claim **14**, further comprising:

a custom module for executing custom virus-detection code responsive to invocation by the engine.

16. The computer program product of claim **14**, wherein the intermediate language is P-code and the engine comprises:

a P-code interpreter for interpreting the P-code and controlling the operation of the engine responsive thereto.

17. The computer program product of claim **16**, wherein the engine further comprises:

primitives for performing operations with respect to the file and the virtual memory responsive to invocations of the primitives by the P-code.

18. The computer program product of claim **14**, further comprising:

a virus definition file for holding virus signatures for use by the scanning module.

19. The computer program product of claim **14**, wherein the instructions stored in the intermediate language post regions of the file for scanning by the scanning module.

20. The computer program product of claim **19**, wherein postings identifying overlapping regions are merged into a single posting identifying the regions of the merged postings.

* * * * *

US005398196A

United States Patent [19]

Chambers

[11] Patent Number: 5,398,196

[45] Date of Patent: Mar. 14, 1995

[54] METHOD AND APPARATUS FOR DETECTION OF COMPUTER VIRUSES

[76] Inventor: **David A. Chambers**, 3655 Eastwood Cir., Santa Clara, Calif. 95054

[21] Appl. No.: **99,368**

[22] Filed: **Jul. 29, 1993**

[51] **Int. Cl.**[6] **G06F 15/20**

[52] **U.S. Cl.** **364/580**; 364/550; 364/578; 364/579; 395/500

[58] **Field of Search** 364/550, 578, 579, 580, 364/286.4; 371/16.2, 19; 395/500

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,080,650	3/1978	Beckett	395/500
4,773,028	9/1988	Tallman	364/578 X
5,086,502	2/1992	Malcom	395/575
5,121,345	6/1992	Lentz	364/550
5,144,660	9/1992	Rose	380/4
5,233,611	8/1993	Triantafyllos et al.	371/19 X

FOREIGN PATENT DOCUMENTS

9006430	4/1991	Brazil .
1057534	1/1992	China .
304033	2/1989	European Pat. Off. .
510244	10/1992	European Pat. Off. .
514815	11/1992	European Pat. Off. .
2629231	9/1989	France .
2632747	12/1989	France .
3736760	5/1989	Germany .
461879	4/1990	Sweden .
2231418	11/1990	United Kingdom .
2246901	2/1992	United Kingdom .
2253511	9/1992	United Kingdom .
9113403	9/1991	WIPO .
9221087	11/1992	WIPO .

OTHER PUBLICATIONS

Bowen, T., "Central Point Tool Unearths New Viruses," *P.C. Week*, May 31, 1993, pp. 41–42.

Excerpts From On-line Documentation for FProt Program, 1994.

Primary Examiner—Edward R. Cosimano

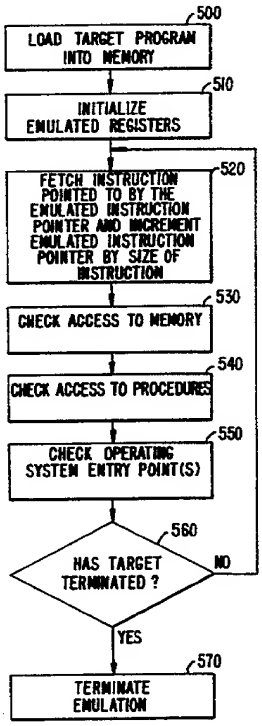
Attorney, Agent, or Firm—Townsend and Townsend Khourie and Crew

[57] **ABSTRACT**

A behavior analyzing antivirus program detects viral infection of a target program by emulating the execution of the target program and analyzing the emulated execution to detect viral behavior. The antivirus monitor program contains both variables corresponding to the CPU's registers and emulation procedures corresponding to the CPU's instructions. The target program is loaded into memory and its execution is emulated by the antivirus monitor program. Intelligent procedures contained in the monitor program are given control between every instruction emulated so as to detect aberrant or dangerous behavior in the target program in which case the danger of a viral presence is flagged and emulation is terminated.

27 Claims, 9 Drawing Sheets

Microfiche Appendix Included
(2 Microfiche, 167 Pages)



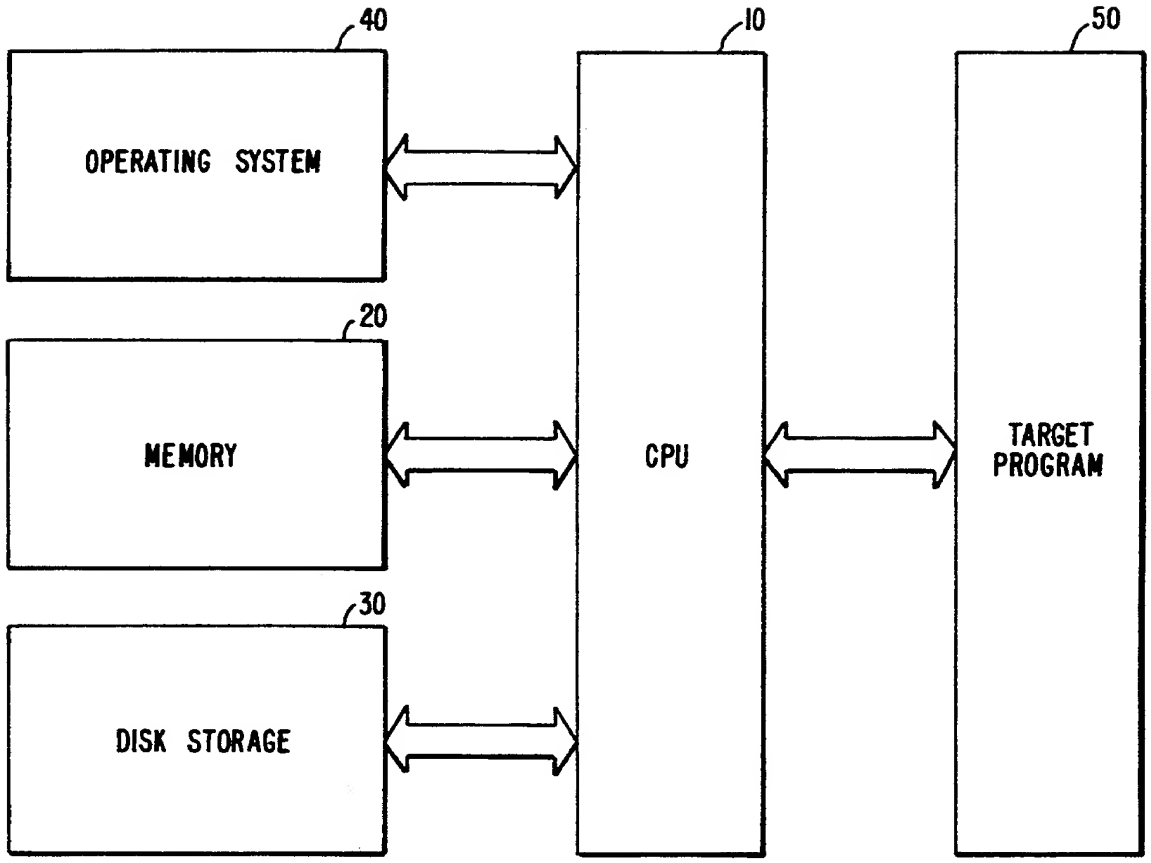


FIG. 1A.
(PRIOR ART)

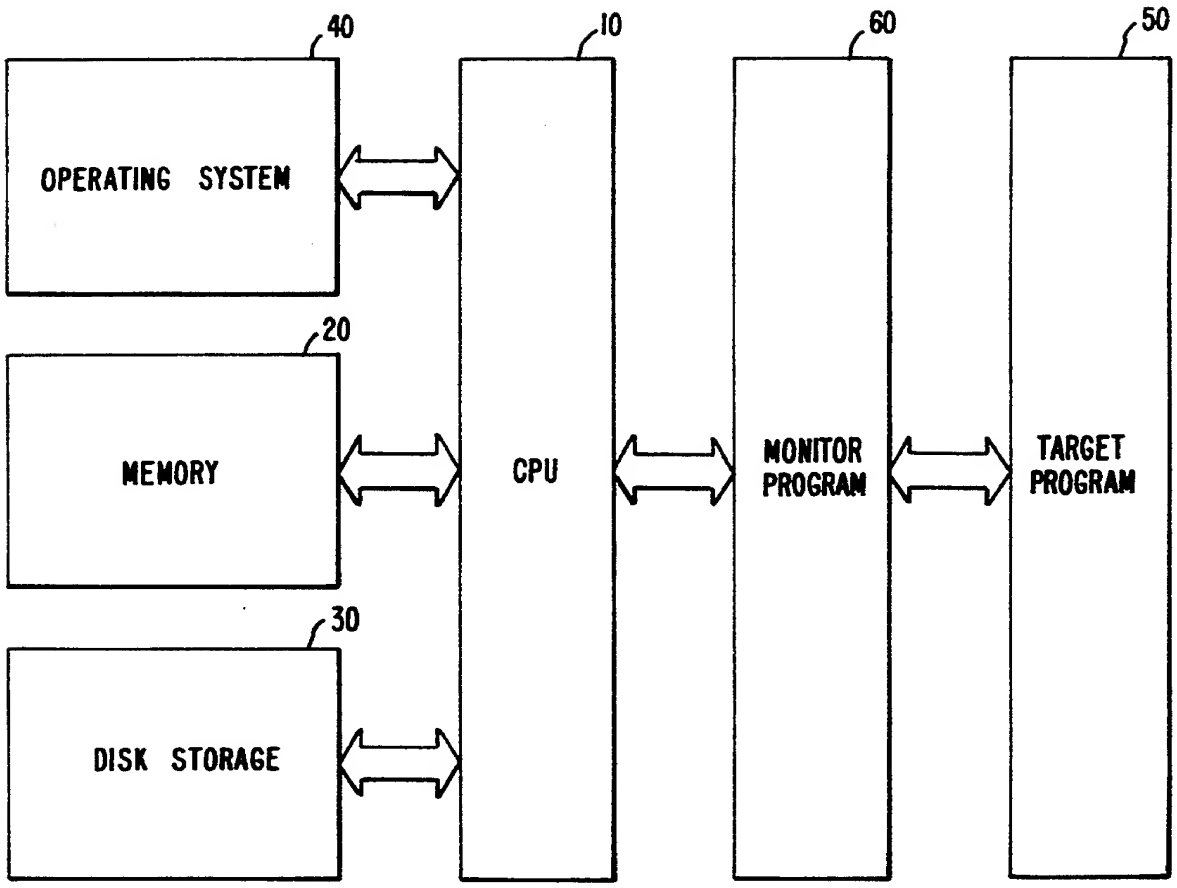

40
10
60
50
OPERATING SYSTEM
20
MEMORY
CPU
MONITOR PROGRAM
TARGET PROGRAM
30
DISK STORAGE

FIG. 1B.

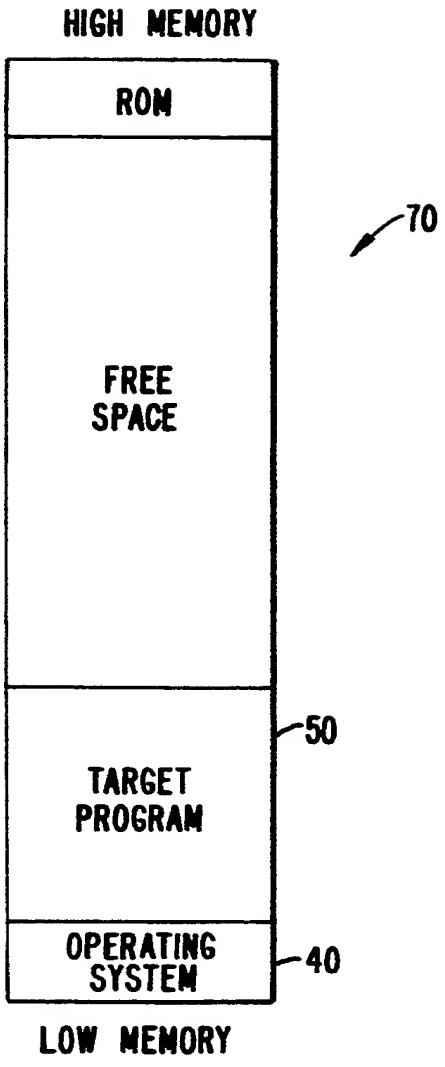


FIG. 2A.
(PRIOR ART)

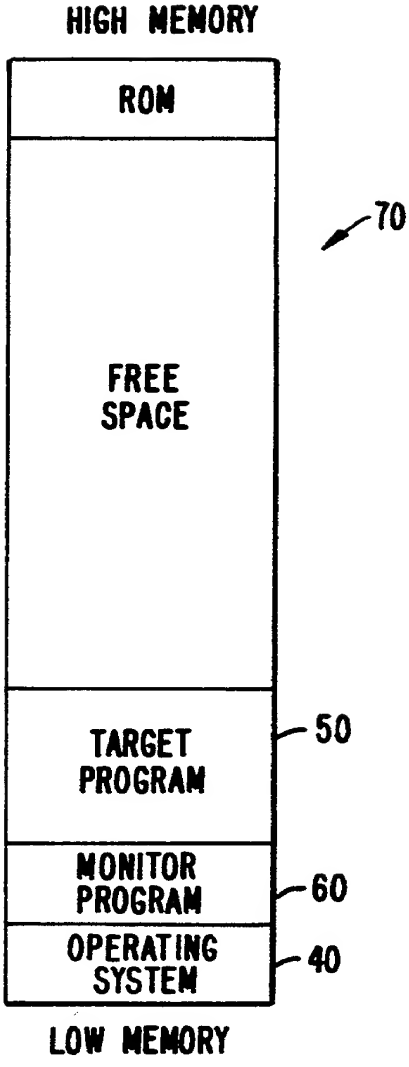


FIG. 2B.

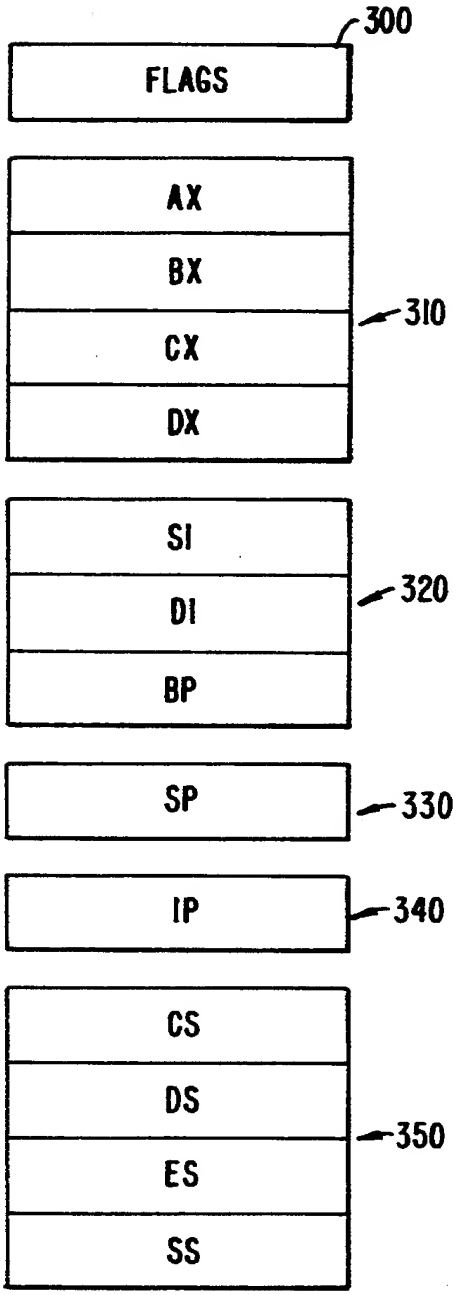


FIG. 3.
(PRIOR ART)

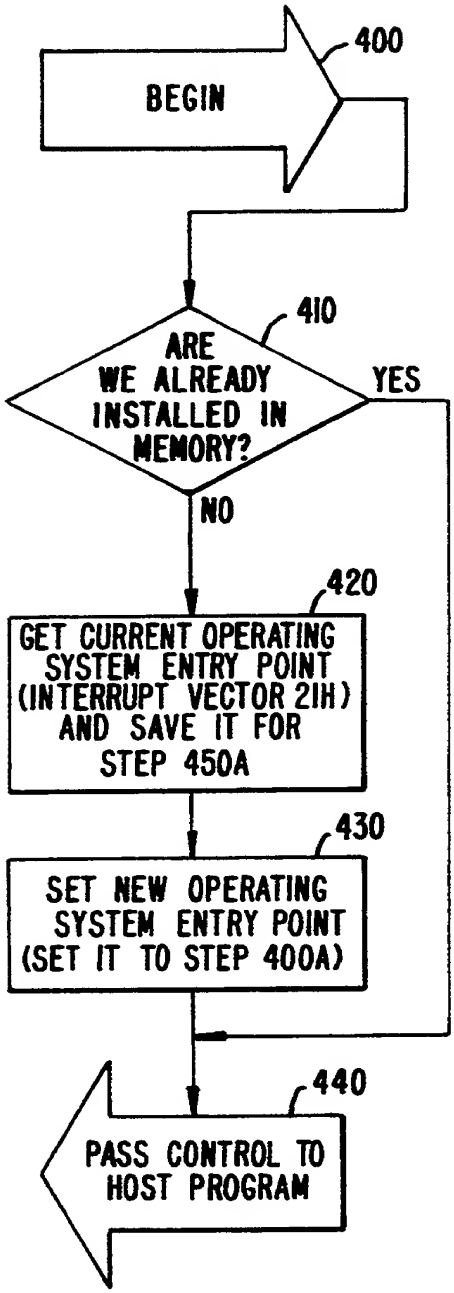


FIG. 4A.
(PRIOR ART)

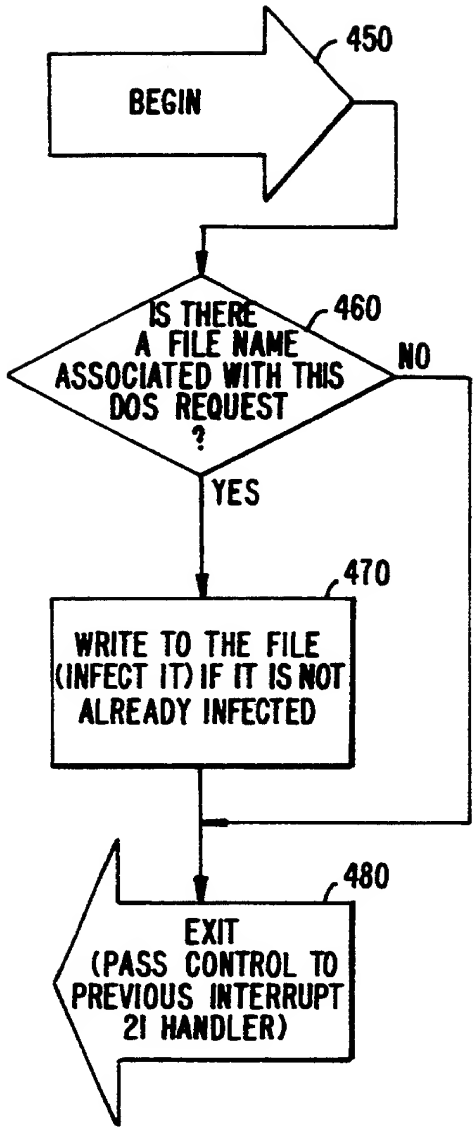


FIG. 4B.
(PRIOR ART)

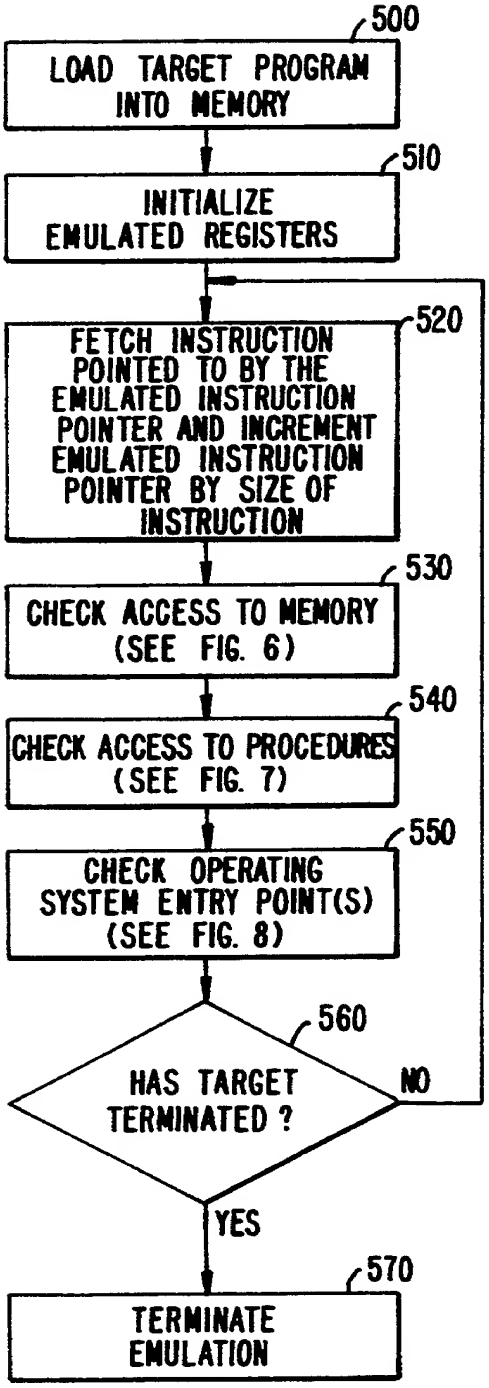


FIG. 5.

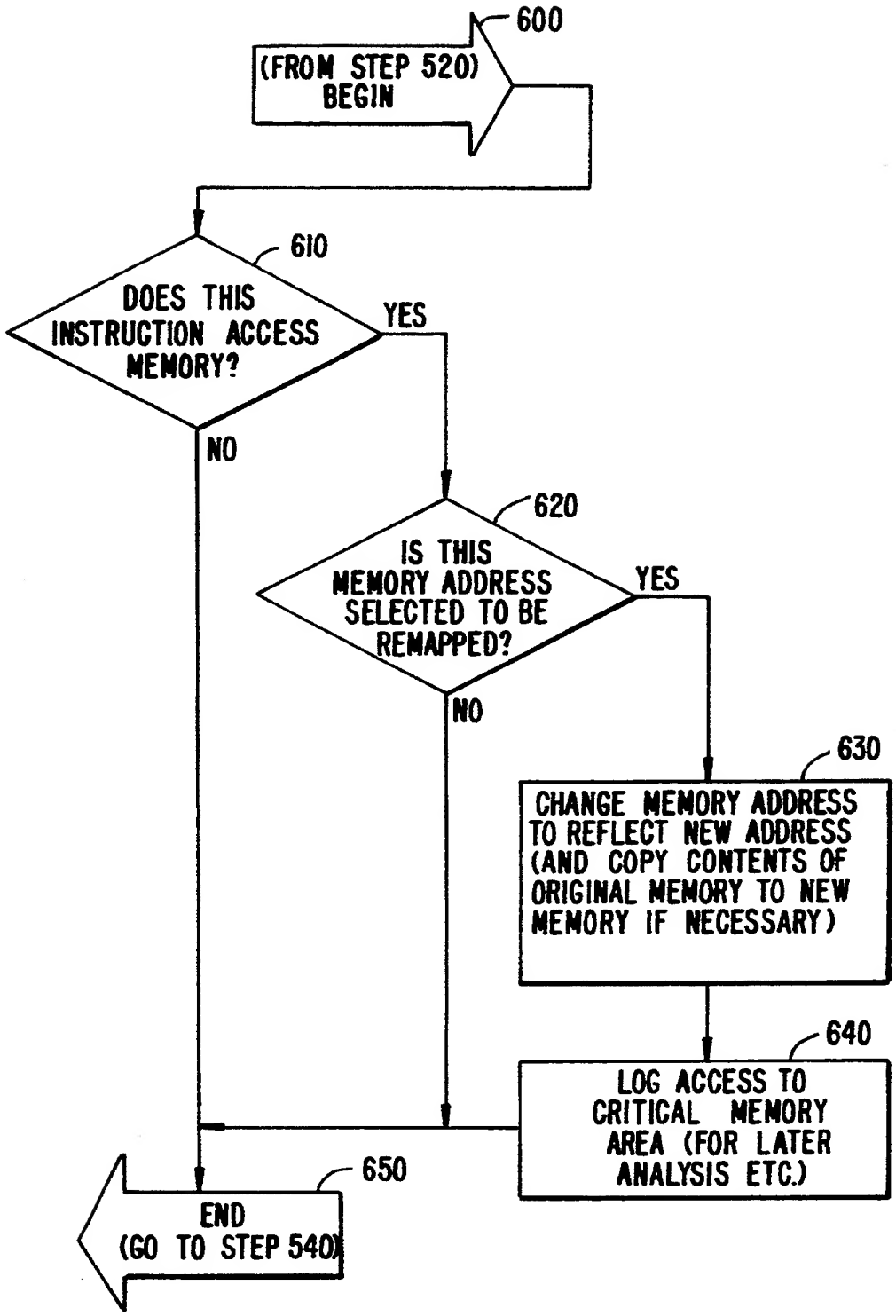

600
(FROM STEP 520)
BEGIN
610
DOES THIS INSTRUCTION ACCESS MEMORY?
YES
NO
620
IS THIS MEMORY ADDRESS SELECTED TO BE REMAPPED?
YES
NO
630
CHANGE MEMORY ADDRESS TO REFLECT NEW ADDRESS (AND COPY CONTENTS OF ORIGINAL MEMORY TO NEW MEMORY IF NECESSARY)
640
LOG ACCESS TO CRITICAL MEMORY AREA (FOR LATER ANALYSIS ETC.)
650
END
(GO TO STEP 540)

FIG. 6.

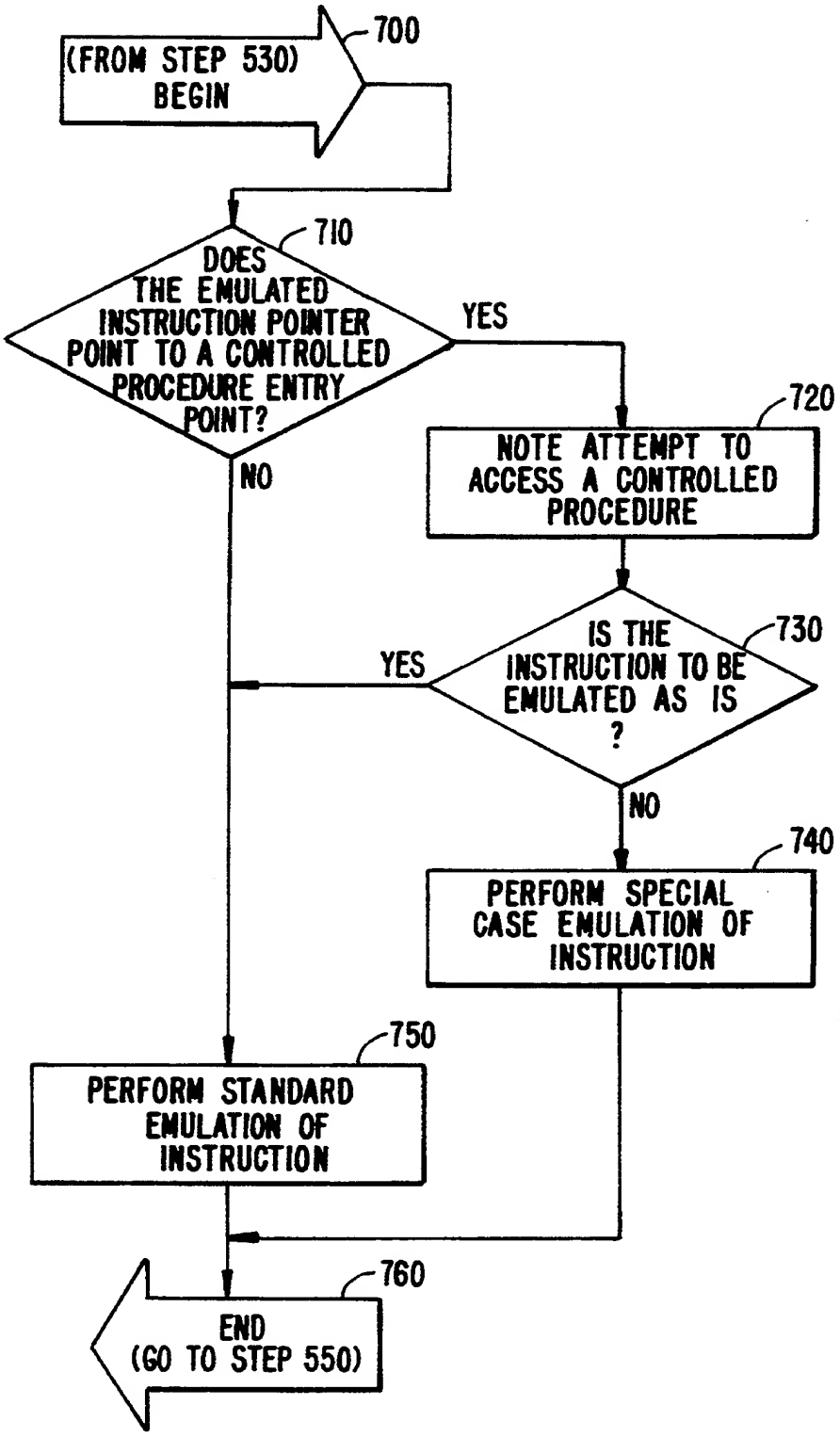

(FROM STEP 530)
BEGIN
700
DOES THE EMULATED INSTRUCTION POINTER POINT TO A CONTROLLED PROCEDURE ENTRY POINT?
710
YES
NO
NOTE ATTEMPT TO ACCESS A CONTROLLED PROCEDURE
720
IS THE INSTRUCTION TO BE EMULATED AS IS ?
730
YES
NO
PERFORM SPECIAL CASE EMULATION OF INSTRUCTION
740
PERFORM STANDARD EMULATION OF INSTRUCTION
750
END
(GO TO STEP 550)
760

FIG. 7.

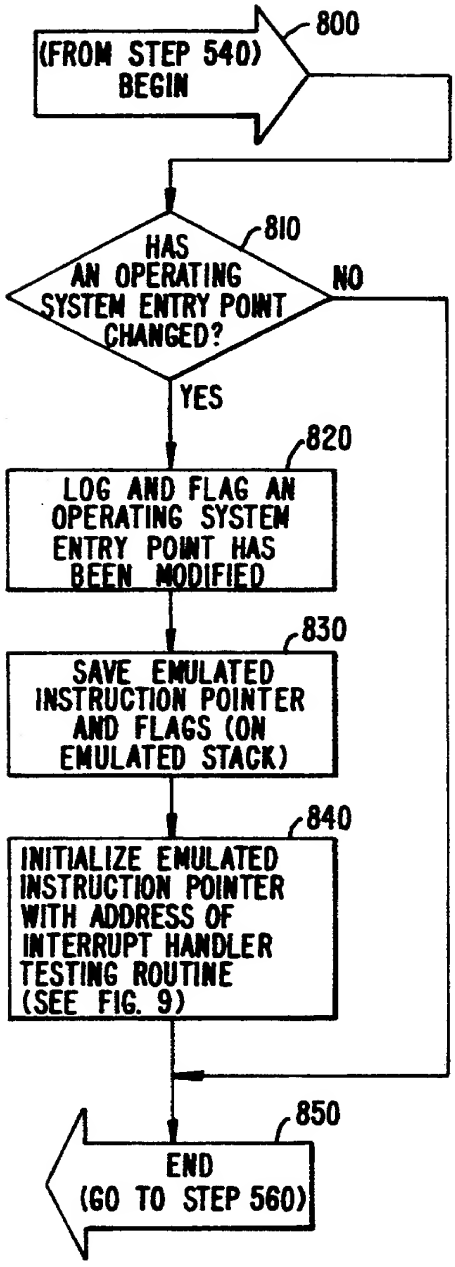

(FROM STEP 540) BEGIN
800
HAS AN OPERATING SYSTEM ENTRY POINT CHANGED?
810
NO
YES
LOG AND FLAG AN OPERATING SYSTEM ENTRY POINT HAS BEEN MODIFIED
820
SAVE EMULATED INSTRUCTION POINTER AND FLAGS (ON EMULATED STACK)
830
INITIALIZE EMULATED INSTRUCTION POINTER WITH ADDRESS OF INTERRUPT HANDLER TESTING ROUTINE (SEE FIG. 9)
840
END (GO TO STEP 560)
850

FIG. 8.

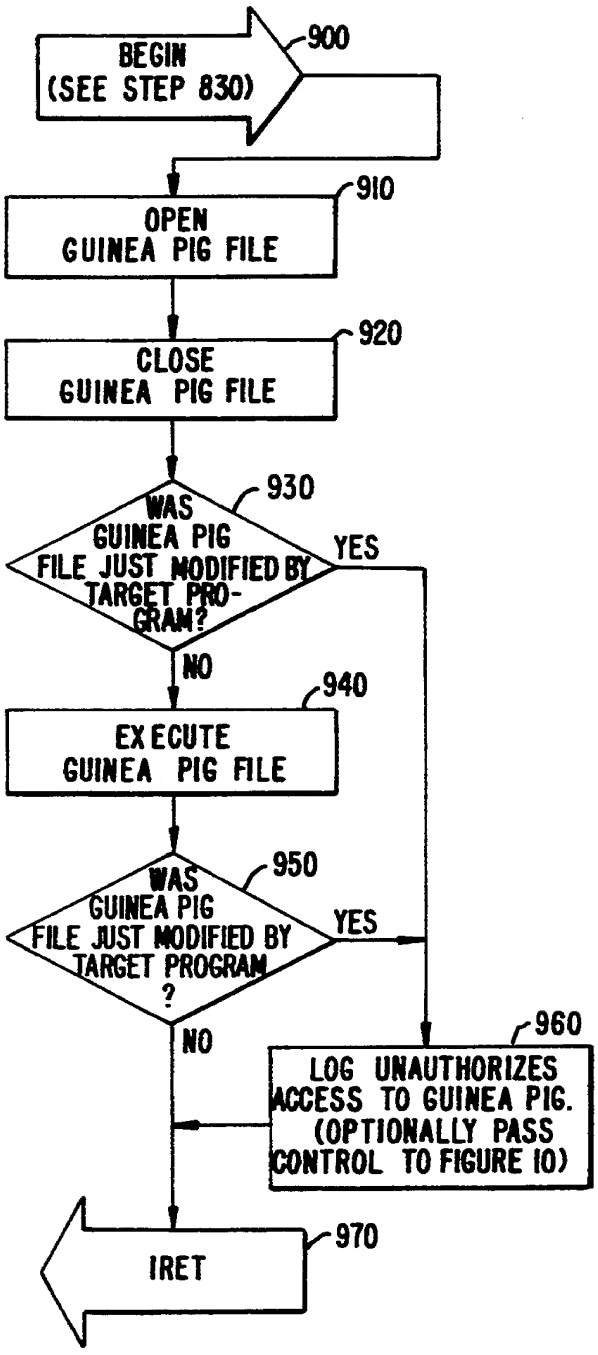

BEGIN (SEE STEP 830)
900
OPEN GUINEA PIG FILE
910
CLOSE GUINEA PIG FILE
920
WAS GUINEA PIG FILE JUST MODIFIED BY TARGET PROGRAM?
930
YES
NO
EXECUTE GUINEA PIG FILE
940
WAS GUINEA PIG FILE JUST MODIFIED BY TARGET PROGRAM ?
950
YES
NO
LOG UNAUTHORIZES ACCESS TO GUINEA PIG. (OPTIONALLY PASS CONTROL TO FIGURE 10)
960
IRET
970

FIG. 9.

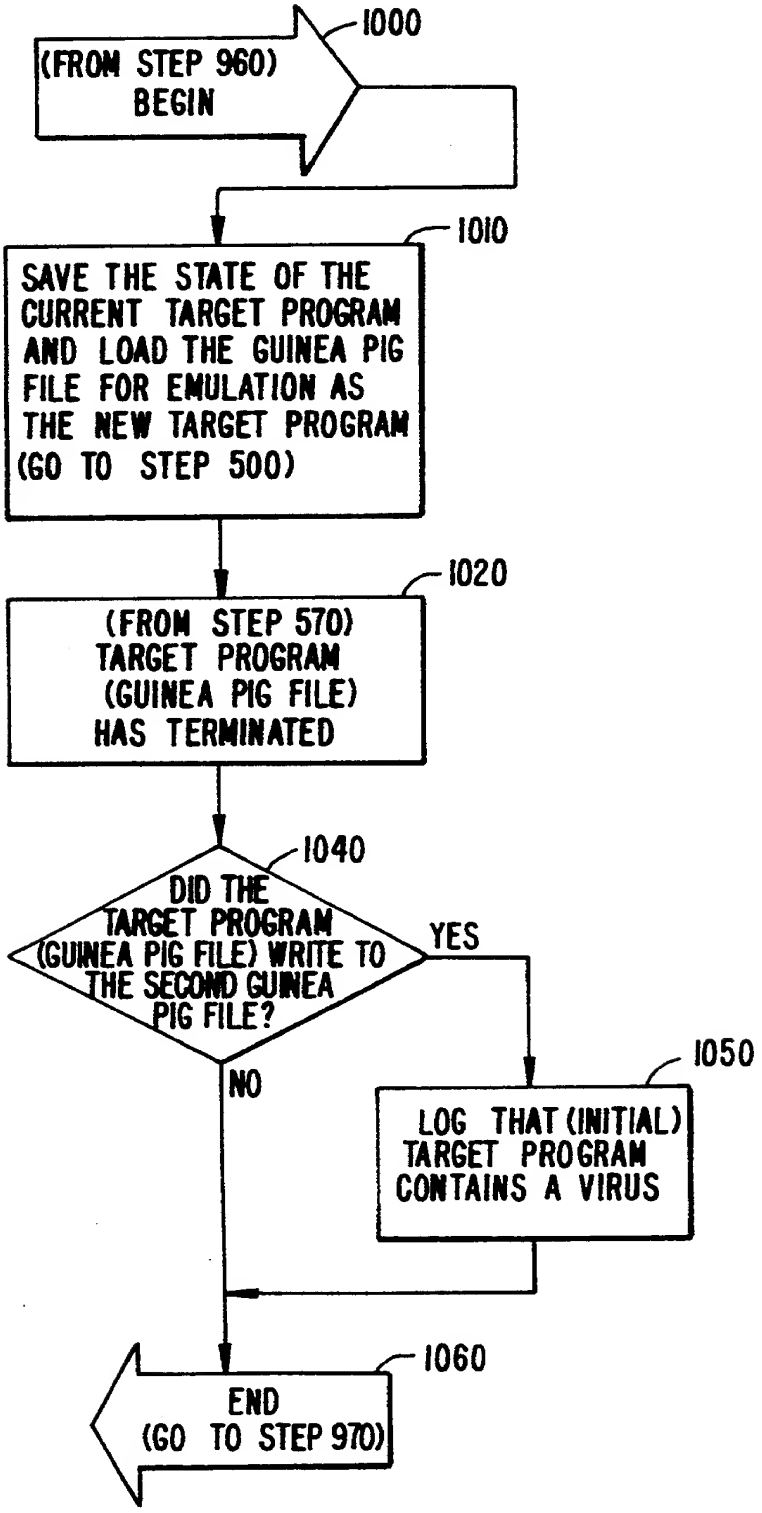

1000
(FROM STEP 960)
BEGIN
1010
SAVE THE STATE OF THE CURRENT TARGET PROGRAM AND LOAD THE GUINEA PIG FILE FOR EMULATION AS THE NEW TARGET PROGRAM (GO TO STEP 500)
1020
(FROM STEP 570) TARGET PROGRAM (GUINEA PIG FILE) HAS TERMINATED
1040
DID THE TARGET PROGRAM (GUINEA PIG FILE) WRITE TO THE SECOND GUINEA PIG FILE?
YES
NO
1050
LOG THAT (INITIAL) TARGET PROGRAM CONTAINS A VIRUS
1060
END
(GO TO STEP 970)

FIG. 10.

METHOD AND APPARATUS FOR DETECTION OF COMPUTER VIRUSES

SOURCE CODE APPENDIX

A microfiche appendix (consisting of 2 sheets and a total of 167 frames) of assembly language source code for a preferred embodiment (©1993 David Chambers) is filed herewith. A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

BACKGROUND OF THE INVENTION

The present invention relates generally to a method and apparatus for emulating the execution of a program on a computer system. In particular, the present invention relates to monitoring program behavior to detect and terminate harmful or dangerous behavior in a program. More particularly, the present invention relates to monitoring program behavior to detect computer viruses.

In recent years, the proliferation of "computer viruses" (generally designed by rogue programmers either maliciously or as "pranks") has become an increasingly significant problem for the owners and users of computer systems. True computer viruses vary, but they share the general characteristic that they comprise executable computer code capable of replicating itself by attachment to and modification of standard computer files. Such files are then considered "infected". On most computer systems, viruses are limited to infecting program applications. When the application is executed, the virus can then replicate and attach copies to further application files. Typically, viruses also engage in other forms of behavior that are considered undesirable, such as re-formatting a hard disk.

Often grouped with true computer viruses are some other types of malevolent computer programs: worms and trojan horses. Worms do not infect other applications but merely replicate, either in memory or in other storage media. The harmful effect of worms is generally to reduce system performance. Worms are of concern for large multiuser computer systems, but are generally not of concern for personal computers. Trojan horses are programs that masquerade as useful programs or utilities; they generally run only once and have a harmful effect (such as destroying or damaging the computer system data storage). Trojan horses do not replicate, and after being run once by a user, the user is usually alerted to the harmful behavior and will not run the trojan horse again.

In response to the proliferation of computer viruses, a variety of "antivirus" methodologies and programs have been developed to detect the presence of infected files. These antivirus programs can be generally categorized into groups: behavior interceptors, signature scanners, and checksum monitors.

BEHAVIOR INTERCEPTORS

The earliest antivirus programs were generally of the behavior interceptor type: they would allow a virus program to execute in memory but would intercept strategic operating system function requests made by the computer virus. Such requests would generally be functions which the virus required to be performed in order to replicate or to destroy its host, i.e., "Write to a file", "Erase a file", "Format a disk" etc. By intercepting these requests, the computer operator/user could be informed that a potentially dangerous function was about to be performed. Control could be halted or continued as necessary. Some antivirus programs actually modify the instructions of the discovered virus program and make them inoperable so as to "kill" them.

The behavior interceptor method of virus detection has several drawbacks. The first problem is that it relies entirely on user input and decision making when potentially dangerous behavior is detected. This places a great burden on the user, for it is often very difficult to determine whether the flagged behavior is part of the normal operation of the program being executed. For example, disk optimizing programs routinely reformat hard disks to improve the interleave value. In response to a warning message, a user might suspect that their disk optimizer was infected with a virus (when in fact it was not) and halt program execution. Or, worse yet, if the user knows that such behavior is part of the normal operation of a disk optimizer program, they would likely allow the format to continue uninterrupted, which would be disastrous if the program were actually infected.

A second problem with behavior interceptor antivirus programs is that computer virus technology has advanced to such a state that some computer viruses are able to bypass the interception points used by the antivirus. The virus can then make operating system function requests that are never intercepted by the antivirus, thus avoiding detection.

A third problem with behavior interceptor antivirus programs is that by allowing the virus to execute, the virus has an opportunity to locate and identify the antivirus program in computer memory. Once the antivirus program is located, the virus can modify the antivirus—rendering it completely ineffective in exactly the same manner that antivirus programs locate and modify virus programs to render them ineffective.

A fourth and very significant problem with behavior interceptor antivirus programs is that there are no low level operating system function requests employed by computer viruses that are not also used by any of thousands of nonvirus programs. At an instruction by instruction level, or at a function-call by function-call level, a computer virus performs the same operations as legitimate computer programs. In other words, the closer a computer virus is examined, the less distinguishable it becomes from any other computer program.

SIGNATURE SCANNERS

The next generation of antivirus technology, signature scanners, answered the problem of over-reliance on user interaction as well as the problem of allowing the virus to execute. A signature scanner operates by knowing exactly what a target virus program code looks like ("signature" code) and then scanning for these program codes in any programs requested to be executed or otherwise requested to be scanned. As long as the signature codes were sufficiently long enough so as not to be confused with another program's code, then positive identification was virtually guaranteed and the request to execute could be stopped before execution ever began. The primary problem with this technique is that it

requires the antivirus developer to have previously collected and analyzed the target viruses, and included the signature codes in the antivirus program. The antivirus program thus relies on an extensive virus signature library, for there are currently several thousand known IBM PC viruses and several new viruses appear each day. Any new viruses appearing after the antivirus program was developed are not included in the library of program codes for which the antivirus can scan. Signature scanning antivirus programs therefore require frequent updates to keep them current with the increasing number of viruses. If the antivirus developer is lax in providing updates, or the user is lax in obtaining and employing available updates, a signature scanning antivirus program can rapidly lose its effectiveness.

CHECKSUM MONITORS

The last standard technique of virus detection does not look for anything to do with viruses in particular, but concentrates on the host programs which the viruses attack. Every program on a system can be "checksummed" at antivirus installation time. Then, when a virus attaches itself to the unsuspecting host program, the checksum value will (probably) be different and the file infected with the virus can be isolated. The primary problem with this technique is that many programs store varying program information within themselves; this will change the checksum value and thus trigger a false alarm virus detection. Another problem is ensuring the integrity of the checksum information, which is typically attached to the program file itself or stored in a separate file. Both locations are vulnerable to covert virus modification. Once a virus infects a host, it can then update the stored checksum value to correspond to the newly infected file and then execute undetected.

SUMMARY OF THE INVENTION

An improved antivirus program according to a first aspect of the present invention avoids the problems of the prior art and detects viral infection of a target program by emulating the execution of the target program and scanning for viral behavior. By emulating the execution of the target program, viruses are prevented from circumventing the monitor program's protective mechanisms. A second aspect of the present invention recognizes that a key viral behavior is replication: viruses generally operate by passing replication/program-modification code onto uninfected programs. Uninfected programs, on the other hand, do not generally add program-modification code to other programs. According to this aspect of the invention, the emulated target program is tested for replication behavior to determine whether the target program is virus-infected.

A monitor program according to the first aspect of the present invention contains both variables corresponding to the CPU's registers and emulation procedures corresponding to the CPU's instructions. The monitor program includes means for loading a target program into memory and emulating its execution. The monitor program also includes means for analyzing the emulated behavior of the target program and for signalling a warning if the emulated behavior is determined to be aberrant, dangerous or otherwise undesirable.

In one embodiment according to the second aspect of the present invention, the monitor program further includes means, responsive to a file access request by the target program, for providing a dummy program, having known behavior, for modification by the target program. The monitor program also has means for emulating the execution of the modified dummy program after the emulation of the target program is complete. If the modified dummy program is determined to have modified functionality, the original target program is flagged as possessing viral behavior. In one particular embodiment according to this aspect of the invention, a first dummy program is known to not possess the ability to modify another file. If after modification by the target program the first dummy program is emulated and found to modify a second dummy program, then the original target program is flagged as virus infected, for having "infected" the first dummy file with aberrant behavior.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1B** is a block diagram illustrating the primary components of a computer system executing a target program in a standard manner.

FIG. **1B** is a block diagram illustrating the primary components of a computer system executing a target program according to the present invention.

FIGS. **2A** and **2B** are diagrams illustrating memory maps of the computer systems of FIGS. **1A** and **1B**, respectively.

FIG. **3** is a block diagram illustrating the register set emulated by a particular embodiment of the present invention.

FIGS. **4A** and **4B** are flowcharts illustrating respectively the installation and replication procedures typically employed by computer viruses.

FIG. **5** is a flowchart illustrating the general emulation process performed by a monitor program according to a particular embodiment of the present invention.

FIG. **6** is a flowchart illustrating in further detail the memory access control step of the flowchart of FIG. **5**.

FIG. **7** is a flowchart illustrating in further detail the procedure access control step of the flowchart of FIG. **5**.

FIG. **8** is a flowchart illustrating in further detail the operating system entry point monitoring step of the flowchart of FIG. **5**.

FIG. **9** is a flowchart illustrating the process performed by a particular embodiment of the present invention to check the behavior of an interrupt handler.

FIG. **10** is a flowchart illustrating the process performed by a particular embodiment of the present invention to identify viral replication behavior.

DESCRIPTION OF THE PREFERRED EMBODIMENT

This description is sufficiently detailed for an understanding of the invention, but for those interested in more details of implementation, a microfiche appendix containing the source code for a particular embodiment is attached. This embodiment is intended for use on IBM PC (or compatible) type computer systems.

In FIG. **1A** a block diagram is shown illustrating the primary components of a computer system executing a target program in a standard manner. The computer system includes a **CPU 10**, a memory **20**, and a disk storage device **30**. This is simply an exemplary configuration; the system could of course employ a tape storage device rather than disk storage, and many other variations are possible as well. Operating system **40** typically exists in Read Only Memory, but may also be partially loaded from the disk storage **30** into memory **20**. At power up, the CPU begins executing the instructions of

operating system **40**, which thereafter controls the loading and execution of application programs such as target program **50**.

In this standard configuration, if a user selects target program **50** for execution, operating system **40** would load target program **50** from disk storage **30** into memory **20** and then transfer control to target program **50** by loading the start address of target program **50** into the program counter register, or instruction pointer register, of CPU **10**. CPU **10** would then begin executing the instructions of target program **50**, as pointed to by the instruction pointer register. Target program **50** will typically include calls to operating system routines, which are identified by a table of pointers, commonly known as interrupt vectors. It is by remapping these interrupt vectors that standard behavior interceptor antivirus programs attempt to maintain control and supervision of target programs. As discussed above, however, many computer viruses are able to circumvent this remapping of the interrupt vectors and are able to use operating system routines without being monitored by the antivirus program.

In order to prevent this circumvention of monitoring code, a particular embodiment of the present invention is invoked by a user to request that an application program be analyzed for viral behavior. This embodiment takes the form of a monitor program that emulates the execution of the application for a period of time, monitoring its behavior. By emulating the execution of the application program, the application program can be maintained in a controlled environment that cannot be circumvented by a virus.

The configuration of the monitor program and target application program is illustrated in FIG. **1B**. Monitor program **60** loads target program **50** into memory and emulates the execution of the instructions of target program **50**, serving as a protective barrier between the application program and the remainder of the computer system. If the application program has not shown any viral behavior at the end of the monitor period, then it is loaded and executed in the standard manner, such as illustrated in FIG. **1A**.

In the secure environment created by the monitor program of FIG. **1B**, every aspect of execution can be scrutinized and the operation of the virus can be controlled completely. If the virus were to request a hard disk format operation, a successfully completed status would be returned to it making the virus "believe" that the operation was successful when in fact it was never executed in the first place.

FIGS. **2A** and **2B** respectively show the general layout in memory **70** for an IBM PC type computer system with a target program loaded directly by PC DOS as in FIG. **1A**, and for a target program loaded by an embodiment of the present invention as in FIG. **1B**. As shown in FIG. **2A**, ROM occupies the upper portion of the memory address space with the remainder of memory being filled up from the bottom: first the operating system **40** in lower memory, followed by device drivers and memory resident programs, then user selected programs such as target program **50**. FIG. **2B** illustrates memory usage as in FIG. **2A**, but additionally with monitor program **60** loaded.

FIG. **3** illustrates the various CPU registers employed by an 8086 type CPU, the general type of CPU employed by many personal computers, and for which the presently described preferred embodiment is intended. Flags register **300** is a set of bit-wise flags the may be set or cleared during the execution of various types of instructions. These bits can be examined by other instructions to alter program flow or to perform other tasks. Registers **310** are general purpose and are used for a variety of tasks. Index registers **320** are typically used to indirectly reference memory. Stack pointer **330** is used to maintain a data storage stack in memory. Instruction pointer (program counter) 340 points to the location in memory at which the next instruction to be executed resides. Finally, segment registers **350** are used to prepend and additional 4 bits onto other memory addressing registers (16 bits wide), allowing them to access a broader range of memory. Because these registers are intimately involved in the execution of programs, they are all emulated by the monitor program of the preferred embodiment, so as to fully control the execution of a target program.

Viral Code

FIG. **4A** illustrates the installation procedure typically employed by computer viruses. The virus execution begins at block **400** and proceeds to block **410**, at which the virus determines if a copy of itself has already been installed in memory. If not, execution precedes to block **420**, where the virus the current value of interrupt vector **21***h* (the operating system entry point on 8086 type computers), and saves this value for later use. Next, at block **430**, the virus sets the entry point to point to a procedure within the virus itself, after which at block **440** control is passed to the host program. If at block **410** the virus had determined that a copy had previously been installed, control would pass immediately to block **440**.

FIG. **4B** illustrates a typical viral procedure for replication. The beginning of such a procedure would be the replacement entry point stored by the viral code at step **430** of FIG. **4A**. When a program later attempts to make an operating system call through int **21**, the call would be directed to beginning block **450** of the viral procedure of FIG. **4B**. The viral .code would then execute, and at block **460** would determine if there was a file name associated with the operating system call. Such operating system calls are typically used by a normal program to open a file or execute another program. If there was a name associated with the operating system call, then at block **470** the viral code would replicate itself by writing its own executable code to the file that was the subject of the operating system call, in some instances after having checked to ensure that this file was not already infected by the virus. After block **470**, the viral code would then exit at block **480**, passing control to the original interrupt handler, a pointer to which had been saved at block **420** of FIG. **4A**. If at block **460** the viral code had determined that there was no filename associated with the operating system call, then execution would have passed directly from block **460** to block **480**. In this manner the operating system continues to function normally except for a slight interruption while the viral code executes.

Emulation to Detect Viral Code

FIG. **5** illustrates the operation of monitor program **60** according to a preferred embodiment of the invention. The monitor program can be executed explicitly by the user with a designated target program, or in alternative embodiments can be executed automatically whenever an operating system call is placed to execute a program. At block **500**, the monitor program loads the

target program into memory, in exactly the same manner as the operating system would have loaded the target program, but rather than passing execution to the target program immediately, the monitor program retains control for a period of time, to evaluate the target program.

After the target program is loaded at block **500**, at block **510** the monitor program initializes the emulated registers, which correspond to the registers used by CPU **10**. These register variables are used by a set of instruction emulation routines that are capable of emulating the instructions of CPU **10**. The emulated registers are initialized with the same values that the real registers would have had if the target program had been loaded by the operating system for execution.

After the emulation registers are initialized, the main emulation loop is entered. At block **520** the instruction pointed to by the emulated program counter register is fetched by the emulation software and the emulated program counter register is incremented by the size of the fetched instruction, so that it points to the next instruction. Control then proceeds to a set of evaluation procedures for the instruction. At block **530**, the monitor program determines if the target program is attempting to access memory selected for controlled access. In the preferred embodiment, operating system procedures and data areas the address range of the monitor program are selected for controlled access. Optionally, any memory not belonging to the target program can be selected for controlled access. The memory access process is explained in more detail below with reference to FIG. **6**.

After block **530**, at block **540** the monitor program evaluates the instruction for attempted access to a controlled procedure, explained more fully below with reference to FIG. **7**, and then also emulates the execution of the instruction. Following block **540** is block **550**, at which the monitor program evaluates any possible modifications to the operating system entry points. The processes performed at block **550** are described in more detail below with reference to FIGS. **8–10**.

Following the emulation and evaluation blocks **5305–50**, at block **560** the monitor code determines if the target application has terminated. If so, emulation is terminated at block **570**. The determination of step **560** can be according to whether the target program terminates of its own accord, or the determination can be set by a total number of instructions to be emulated or by a fixed period of time for emulation. If the target program has not terminated of its own accord at step **560**, and if the monitor program has not forcibly terminated it, control returns to block **520**, where the next cycle of the emulation loop is begun. The emulation termination at block **570** includes some "cleanup" on the part of the monitor program. This includes displaying to the user a status report of all operating system requests performed by the target program. This step may optionally also include reporting any memory accesses that have been performed outside of the area provided for the target program by the monitor program.

Controlling Access to Memory

The memory access monitoring process of block **530** is illustrated in further detail in FIG. **6**. The described process involves remapping selected parts of memory, which effectively virtualizes those memory areas, making them inaccessible to the target program, and thus protected. In alternative embodiments, access by the target program to these areas of memory is simply denied by the monitor program.

From the starting point at block **600**, the procedure passes to block **610**, at which the monitor program determines if the current instruction is one whose function is to access memory. If so, then control passes to block **620**, where the monitor program determines if the memory location to be accessed by the current instruction is in an area selected for controlled access. If so, then control passes to block **630**, which implements a remapping of the memory address. The monitor program's representation of the instruction is modified to point to the mapping destination, so that the original memory location is protected from the target program.

In the preferred embodiment, the contents of the original memory location are copied to the mapping destination the first time the location is accessed by the target program. In other embodiments, the contents of the entire memory area selected for controlled access are copied into the mapping destination area when the monitor program first starts. In yet other embodiments, certain areas selected for controlled access can have their mapping destination areas initialized with null or dummy values. For example, it may be desirable that the content of the monitor program be protected and hidden from the target program, so that a virus cannot detect the presence of the monitor program.

After the remapping of block **630**, at block **640** the attempted access to a controlled memory area is logged for later analysis and reporting to the user. After block **640**, the memory access control procedure ends at block **650**, which returns control to the main process of FIG. **5**, at block **540**. A negative determination at either of blocks **610** or **620** also results in control passing immediately to block **650**.

Controlling Access to Procedures

In some instances, it is desirable to control access to certain procedures. For instance, operating system procedures, ROM procedures, and interrupt handling procedures can have powerful effects and can be subject to misuse by a virus. For these reasons, it is desirable to control access to them and substitute special purpose procedures in their place, to encapsulate viral code within the emulated environment.

After the memory access control procedure of block **530** of FIG. **5**, control passes to block **540**, which is illustrated in further detail in FIG. **7**. From beginning block **700** control passes to block **710**, at which the monitor program determines if the emulated program counter points to a controlled procedure entry point; a list of such entry points is maintained by the monitor program. If so, then at block **720** the attempted access to a controlled procedure is noted. This can be by displaying a message to the user on the screen, writing to a log file, etc.

Next, at block **730** the monitor program determines if the instruction is to be directly emulated. This determination is made according to information stored for each controlled procedure entry point; for certain such procedures a special case emulation may be desired rather than directly emulating the instructions of the procedure. If the procedure is not to be directly emulated, then control passes to block **740**, where a special case emulation of the entry point instruction is performed. In some instances this special case emulation will entail emulation of the entire controlled procedure at this point.

If at block **730** it were determined that the controlled procedure was to be directly emulated, or if at block **710** it were determined that the emulated instruction pointer did not indicate a controlled procedure entry point, then control would pass to block **750**, where the instruction indicated by the emulated instruction pointer is emulated in the same manner as other instructions. Following the emulation according to either of blocks **740** or **750**, control passes to block **760**, which returns execution to the main process of FIG. **5**

Controlling Access to Operating System Entry Points

Block **550** is illustrated in further detail by FIG. **8**. This control of operating system entry points need not be performed to obtain substantial benefits from the emulation of the target program; however, this process does a higher level of control over the target program and also allows for a more accurate evaluation of viral behavior on the part of the target program.

From beginning block **800** control passes to block **810**, at which the monitor program examines a list of operating system entry points to determine if any have changed as a result of the instruction just emulated. This would indicate that the target program had replaced an interrupt handler with a routine of its own. If there is such a change, then it is logged at block **820**. At block **820** a flag is also preferable set to indicate that the entry point has changed, so that the change will not be logged redundantly later. In some embodiments, the flag indicates the new value of the entry point, so the monitor program can determine if the entry point gets modified yet again.

After block **820**, at block **830** the emulated instruction pointer, emulated code segment register, and emulated flag register are saved onto the emulated stack. Then the emulated stack pointer is decremented the corresponding 6 bytes, in the same manner as if a hardware interrupt had been received. Next, at block **840**, the emulated code segment register and emulated instruction pointer are set to a special purpose monitor program routine to test the interrupt handler just installed by the target program. This interrupt handler testing routine is described below with reference to FIG. **9**.

After block **840**, execution passes to block **850**, which returns control to the basic process of FIG. **5**. This causes the interrupt handler routine of FIG. **9** to be emulated in the same step by step manner as the target program. This maintains the highest degree of encapsulation around the target program, although if detecting viral replication is essentially the only concern, the interrupt handler testing routine of FIG. **9** may alternatively be executed in a more straightforward emulation without many of the execution safeguards described above.

If at block **810** the monitor program had determined that no operating system entry points had been changed, then control would have passed directly to block **850**, and thus returned to the process of FIG. **5** to emulate the remainder of the target program.

Interrupt Handler Testing

The basic tack of the interrupt handler testing routine is to offer up a guinea pig file for "sacrifice" to a potential viral interrupt handler, and then test the guinea pig file for corruption. This requires that a "clean" guinea pig file already be at hand and also be disposable. This can be easily provided for by several methods, such as by creating the guinea pig file or copying the guinea pig file from a clean library copy at the very start of the monitor program. The guinea pig file should have a known content. It is preferably executable, but without its execution involving writing to other files. The guinea pig file can thus be essentially a null file that does nothing when executed, simply returning immediately.

As shown in FIG. **9**, when the interrupt handler routine is entered at block **900**, the first action is to open the guinea pig file, at block **910**, after which the guinea pig file is closed at block **920**. Next, at block **930** the interrupt handler testing routine examines the guinea pig file to determine if its content has been changed. Such would be the result of a virus having contaminated the interrupt handlers for opening or closing files. If a change is not detected at block **930**, then at block **940** the guinea pig file is executed, after which at block **950** the guinea pig file is again examined by the interrupt handler testing routine to determine if its content has been changed by the execution interrupt handler.

If a positive determination had been made at either of blocks **930** or **950**, then execution would pass from the respective block to block **960**, at which the unauthorized access to the guinea pig file would be logged. After block **960**, and also after a negative determination at block **950**, execution passes to block **970**, which executes an (emulated) IRET instruction. This is a return from interrupt instruction, which causes the values placed onto the emulated stack at block **830** of FIG. **8** to be restored to the emulated registers. This completes the interrupt handler testing, and returns the emulation to its last point of emulation in the target program.

For a more refined and definitive degree of analysis, block **960** can also initiate a routine to determine not just if the guinea pig was contaminated, but if it was contaminated in a way so as to contaminate other files; i.e., if it was infected with viral replication behavior. Such a routine is illustrated in FIG. **10**. The process of FIG. **10** essentially creates a completely new emulation, with the modified guinea pig file serving as the target program. If this first guinea pig file now passes modification behavior on to a second guinea pig file, then the original target program has been shown to be contaminated with viral code having replicative behavior. To prevent needless additional recursion, the second level of emulation should be identified as such, through use of a flag, etc., so that if block **960** is reached during the second level of emulation, viral behavior is confirmed and the second level of emulation is terminated (rather than beginning another level of testing with yet another guinea pig file).

This replication detection process is illustrated in FIG. **10**. After beginning at block **1000**, at block **1010** the complete state of the current emulation is saved, and all operating system entry points, etc., are returned to their values at the beginning of the first emulation. Block **1010** also then includes the step of initiating emulation again, but with the guinea pig file specified as the target program. As noted above, this emulation level should be flagged as a second level emulation.

Block **1020** indicates the point at which the emulation of the guinea pig file has terminated, after which at block **1040** the first level emulation determines if the emulated guinea pig file had written to a second guinea pig file. This determination is most straightforward if a flag is simply passed from the second level emulation back to the first; it can also be by examining a checksum for the file. If the determination at block **1040** is positive, then at block **1050** the initial target program is

confirmed and logged as being virus-contaminated. contaminated. After either a negative determination at block **1040** or execution of block **1050**, execution proceeds to block **1060**. At block **1060** at the end of the process of FIG. **10**, control is passed back to block **970** of FIG. **9**, to continue emulation of the initial target program. Alternatively, reaching block **1050** can result in the entire emulation being terminated, as the target program has been confirmed as being virus-contaminated.

Source Code Appendix

The embodiment of the source code appendix is intended for use on IBM PC type computer systems. The source code is in the form of a number of 80×86 assembly language files that must be compiled and then linked in the order VPROBE, CPU, PROT, VECTORS, DIAG, DEBUG, THEEND. The resultant executable file can then be executed with a program name, passed as a command line parameter, to serve as a target program for virus detection.

Alternative Embodiments

Rather than requiring the user to load the monitor program which then loads the target program, a "zero length loader" TSR version could be installed in a system and every program requested to be executed could be emulated. If no abnormal behavior is found in the first 'n' instructions, the monitor program could pass control to the CPU to allow the target to execute at "full speed" and the end user would not have to be aware of the existence of the monitor program (other than a slight delay during the initial execution).

Another alternative approach would be where a recursive parser/emulator could effectively evaluate every single instruction of executable code in a program by noting the address of conditional branch instructions, and returning to that branch location, restoring the cpu/memory state of the machine at that instant, and continuing emulation as if the branch had taken the alternate route instead. Emulation continues until all instructions have been evaluated. This would be a time consuming process; however, the information revealed would definitively answer the question of whether the original code was virus infected.

It is also important to note that, although the described embodiment is oriented towards identifying viral behavior, the disclosed emulation techniques can be constructively employed to emulate program execution in all types of situations where potentially destructive or other predetermined program behavior is a concern.

It is to be understood that the above description is intended to be illustrative and not restrictive. Many other embodiments will be apparent to those of skill in the art upon reviewing the above description. For instance, the instructions of the emulated application program could be read directly from disk storage rather than being loaded into memory first. The scope of the invention should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.

What is claimed is:

1. A computer system configured to monitor the execution of a target program intended to run on a target computer, said target computer having an instruction set, said computer system comprising:

a processing unit;

instruction emulation means for operating said processing unit to emulate instructions in said target program corresponding to said instruction set as if said instructions were running on said target computer;

monitor means, coupled to said instruction emulation means, for emulating execution of said target program as if said target program were running on said target computer and for monitoring said emulated target program execution to detect a predetermined behavior by said target program; and

means, coupled to said monitor means, for logging said predetermined behavior when detected.

2. The computer system of claim **1**, wherein said computer system is configured to detect viral activity associated with said target program, wherein said predetermined behavior is chosen to be indicative of effects of said viral activity.

3. The computer system of claim **1** wherein said instruction emulation means comprises:

a register emulator for emulating registers of said target computer; and

a procedure controller for substituting emulation procedures for procedures accessed by said target program during emulation.

4. The computer system of claim **1** wherein said instruction emulation means comprises:

a memory access controller for controlling access by said instructions to memory.

5. The computer system of claim **1** wherein said target computer is said computer system.

6. In a computer system, a method for monitoring execution of a target program intended to run on a target computer comprising the steps of:

emulating the target program as if it were running on said target computer; and

monitoring emulation of the target program to detect a predetermined behavior indicating presence of a computer virus.

7. The method of claim **6** wherein said target computer is said computer system.

8. In a computer system including a processing unit and a viral behavior monitor, a method for monitoring execution of a target program intended to run on a target computer comprising the steps of:

using said processing unit to emulate execution of an instruction as if said instruction were running on said target computer;

using said viral behavior monitor to control access by said instruction to memory;

using said viral behavior monitor to control access by said instruction to procedures; and

repeating said step of using said processing unit for successive instructions of the target program.

9. The method of claim **8** wherein said step of using said viral behavior monitor to control access by the instruction to memory comprises the substeps of:

substituting an alternative memory address for a memory address to which the instruction intends access and which has been preselected for controlled access; and

logging an attempted access to the memory address to which the instruction intends access.

10. The method of claim **8** wherein said target computer is said computer system.

11. The method of claim **8** wherein said step of using said vital behavior monitor to control access by the instruction to procedures comprises the substep of:

upon a determination that the instruction is located at a controlled procedure entry point, logging an access attempt to the controlled procedure entry point.

12. The method of claim **11** wherein said step of using said viral behavior monitor to control access by the instruction to procedures further comprises the substeps of:

upon a determination that the controlled procedure entry point belongs to a special case list, performing a special case emulation of the instruction; and

upon a determination that the controlled procedure entry point does not belong to a special case list, performing a standard emulation of the instruction.

13. The method of claim **12** wherein said step of performing a special case emulation of the instruction comprises substituting an alternate procedure for the procedure which begins at the controlled procedure entry point.

14. The method of claim **8** further comprising the step of:

using said viral behavior monitor to control access by the instruction to operating system access points.

15. The method of claim **14** wherein said step of using said vital behavior monitor to control access by the instruction to operating system access points comprises the substep of:

upon a determination that an operating system entry point has changed, logging an operating system entry point change.

16. The method of claim **15** wherein said step of using said viral behavior monitor to control access by the instruction to operating system access points further comprises the substeps of:

opening a guinea pig file;

closing the guinea pig file;

examining the guinea pig file to check if contents of the guinea pig file have changed; and

upon a determination that contents of the guinea pig file have changed, logging unauthorized access to the guinea pig file.

17. The method of claim **15** wherein said step of using said viral behavior monitor to control access by the instruction to operating system access points comprises the substeps of:

executing a guinea pig file;

examining the guinea pig file to check if contents of the guinea pig file have changed; and

upon a determination that contents of the guinea pig file have changed, logging unauthorized access to the guinea pig file.

18. The method of claim **16** wherein said step of using said viral behavior monitor to control access to operating system entry points further comprises the substeps of:

upon a determination that contents of the guinea pig file have changed, loading the guinea pig file for emulation as a new target program and proceeding through said steps of emulating execution and controlling access to operating system entry points, thereby creating a second guinea pig file; and

upon a determination that the new target program has written to the second guinea pig file, logging a virus.

19. The method of claim **17** wherein said step of using said viral behavior monitor to control access to operating system entry points further comprises the substeps of:

upon a determination that contents of the guinea pig file have changed, loading the guinea pig file for emulation as a new target program and proceeding through said steps of emulating execution and controlling access to operating system entry points, thereby creating a second guinea pig file; and

upon a determination that the new target program has written to the second guinea pig file, logging a replicative virus.

20. A computer system configured to monitor the execution of a target program intended to run on a target computer, said target computer having an instruction set, said computer system comprising:

a processing unit;

an instruction emulator for operating said processing unit to emulate instructions corresponding to the instruction set as if said instructions were executed by said target computer;

an entry point access controller for controlling access to operating system entry points; and

a logger for logging improper access by said instructions to operating system entry points.

21. The computer system of claim **20** further comprising a procedure access controller for controlling access to procedures during instruction emulation, wherein said logger logs improper access by said instructions to procedures.

22. The computer system of claim **20** further comprising a memory access controller for controlling access by said instructions to memory during instruction emulation wherein said logger logs improper access by said instructions to memory.

23. The computer system of claim **20** wherein said target computer is said computer system.

24. The computer system of claim **20** wherein said entry point access controller checks if a viral interrupt has been installed.

25. The computer system of claim **24** wherein said entry point access controller opens and closes a guinea pig file and tests for modification of the guinea pig file to determine if a viral interrupt has been installed.

26. The computer system of claim **24** wherein said entry point access controller executes a guinea pig file and tests for modification of the guinea pig file to determine if a viral interrupt has been installed.

27. The computer system of claim **26** wherein said entry point access controller executes the guinea pig file as a new target program, thereby creating a second guinea pig file, and tests for modification of the second guinea pig file to determine if a replicative viral interrupt has been installed.

* * * * *

APPENDIX C

Related Proceedings Appendix

As stated on page 3 of this Appeal Brief, to the knowledge of Appellants' counsel, there are no known appeals, interferences, or judicial proceedings that will directly affect or be directly affected by or have a bearing on the Board's decision regarding this Appeal.